Libros de Cátedra

Lógica para informática

Claudia Pons, Ricardo Rosenfeld y Clara Smith

e

exactas

FACULTAD DE
INFORMÁTICA





LÓGICA PARA INFORMÁTICA

Claudia Pons
Ricardo Rosenfeld
Clara Smith

Facultad de Informática





Agradecimientos

Los autores agradecemos a las autoridades de la Universidad Nacional de La Plata (UNLP) por permitirnos dar a luz esta publicación.

Al Prof. Gabriel Baum: su prólogo a nuestro libro nos llena de orgullo, máxime por quien lo ha elaborado.

A nuestras familias, por tolerar nuestras ausencias durante este trabajo con infinita paciencia y fundamentalmente por su aliento constante.

Y por supuesto, y de especial manera, a nuestros estudiantes, quienes a lo largo de los años han contribuido con sus consultas y opiniones a la apropiada presentación de los temas que integran *Lógica para Informática*.

Además:

Clara Smith agradece el apoyo y financiamiento recibidos del programa de investigación e innovación Horizon 2020 de la Unión Europea bajo el acuerdo Marie Sklodowska Curie N° 690974 para el proyecto "MIREL: MIning and REasoning with Legal texts", y del proyecto de investigación "Responsabilidad en Sistemas Multiagente" Re N° 1156/14 de la Universidad Católica de La Plata, 2014. También agradece al Dr. Matías Menni por las discusiones y comentarios sobre el Capítulo 3 de este libro.

Claudia Pons y Ricardo Rosenfeld recuerdan y agradecen con especial cariño y respeto a César Archuby, Armando Haeberer y Jorge Boria, SIEMPRE presentes. A comienzos de los años 1980 impulsaron el funcionamiento del Grupo FUNDAPRO (por Fundamentos de la Programación), en el marco de la carrera de Calculista Científico de la Facultad de Ciencias Exactas de la UNLP. Eran los años en que llegaban a la región, "fresquitos", libros fundacionales acerca de las metodologías de programación y la verificación formal de programas (Dijkstra, Jackson, Liskov, etc). Nosotros los "devorábamos", influenciados por el carisma de nuestros mentores. Un poco después siguieron la Escuela Superior Latinoamericana de Informática (ESLAI) y la Escuela Brasilero-Argentina de Informática (EBAI), consecuencias naturales de iniciativas como la mencionada, e impulsadas por los mismos y más actores, en las que tuvimos la inmensa suerte de participar. Todo ello nos dejó una huella imborrable en nuestras vidas, que ojalá se vea reflejada en las hojas de este libro.

Índice

Prólogo

Escribir un buen libro de texto sobre lógica para informática es una contribución difícil de sobreestimar. Entre tanto ruido de nombres de tecnologías, lenguajes, dialectos, normas, estándares, y demás deudos del gigantesco negocio de las TICs, presentar a los estudiantes los fundamentos últimos de todo el paquete de conocimientos que posibilitan la construcción de la tecnología debe saludarse, sobre todo cuando se hace bien, con simpleza y claridad, sin perder profundidad ni rigor. Tal es el caso de este gran material.

Destino de los buenos libros, que deben leerse con cuidado, tiempo y constancia. No despiertan enormes entusiasmos, no hay muchos promotores de estos asuntos. Por una parte, el establishment académico mundial viene empujando hacia el desván de las cosas viejas el aprendizaje de las ciencias "duras" y los formalismos; el “Consenso de Bologna” es un ejemplo de estas prácticas. Gran parte de los "demandantes de recursos humanos", las cúpulas empresarias, exigen empleados listos para usar, sin buenas ideas o intereses “exóticos”. Las políticas educativas concilian ambas demandas, y le suman sus propias necesidades de índices de “calidad educativa”, esto es, buena tasa ingreso/egreso sin muchas exigencias de aprendizajes profundos y relevantes. A pocos les importa el desafío fundamental de la Universidad: preparar personas con pensamiento crítico, capaces de enfrentar los desafíos de la sociedad en los próximos 50 años. Un acierto de la Editorial de la UNLP, publicar *Lógica para Informática*.

Hace unos 30 años, E. Dijkstra, el pensador más lúcido y profundo de las ciencias de la computación desde la segunda mitad del siglo XX, escribió un brillante artículo llamado *On the cruelty of really teaching computer science* (es decir, *Acerca* de la crueldad de *realmente enseñar ciencias de la computación*). Allí, Dijkstra reflexiona acerca de las dificultades para comprender qué es la computación y para qué enseñarla, en particular enseñar a programar. Las dificultades provienen de la propia naturaleza de la computación. La computadora digital moderna introduce dos novedades radicales:

“La primera novedad radical es una consecuencia directa del poder brutal de las computadoras actuales. Todos sabemos cómo lidiar con algo grande y complejo: divide y vencerás, vemos el todo como una composición de partes y tratamos con las partes por separado. Y si una parte es muy grande, repetimos el procedimiento. La ciudad está compuesta por barrios, que están a su vez estructurados por calles, que contienen edificios, que están hechos de paredes y pisos, que están construidos de ladrillos, etc., eventualmente llegando a las partículas elementales. Y tenemos a todos nuestros especialistas sobre el tema, desde el ingeniero civil, pasando por el arquitecto hasta el físico de estado sólido y consiguientes. Dado que, en cierto sentido, el todo es "más grande" que

sus partes, la profundidad de una descomposición jerárquica es algún tipo de logaritmo del cociente entre los "tamaños" del todo y las partes más pequeñas. Desde un bit a cien mega bytes, desde un microsegundo a media hora de cómputos, nos enfrentamos con un cociente completamente abrumador de ¡10^9! El programador está en una posición inigualada, la suya es la única disciplina y profesión donde un cociente tan gigante, que sobrepasa completamente nuestra imaginación, debe ser consolidado por una sola tecnología. Debe poder pensar en términos de jerarquías conceptuales que son mucho más profundas que todas aquéllas que debió enfrentar una sola mente alguna vez."

Más adelante indica Dijkstra:

"La segunda novedad radical es que la computadora automática es nuestro primer dispositivo digital de gran escala. Tuvimos un par de notables componentes discretos. Acabo de mencionar la caja registradora y podemos agregar la máquina de escribir con sus teclas individuales: con un solo golpe podemos escribir una Q o una W pero, aunque las teclas están una al lado de la otra, no hay una mezcla de las dos. Pero tales mecanismos son la excepción, la amplia mayoría de nuestros mecanismos son vistos como dispositivos analógicos cuyo comportamiento sobre un amplio rango es una función continua de todos los parámetros involucrados: si presionamos la punta del lápiz un poco más fuerte, obtenemos una línea levemente más gruesa, si el violinista ubica su dedo levemente fuera de su posición correcta, reproduce una nota levemente desafinada. A esto debería agregar que, en tanto nos vemos a nosotros mismos como mecanismos, nos vemos primordialmente como dispositivos analógicos: si nos esforzamos un poco más esperamos rendir un poco más... Es posible, inclusive tentador, ver a un programa como un mecanismo abstracto, como alguna clase de dispositivo. Pero hacerlo, sin embargo, es altamente peligroso: la analogía es muy superficial debido a que un programa es, como mecanismo, totalmente diferente de todos los familiares dispositivos analógicos con los cuáles crecimos. Como toda la información digitalizada, tiene la inevitable e incómoda propiedad de que la menor de las posibles perturbaciones - por ejemplo cambiar un solo bit - puede tener las más drásticas consecuencias. (Por completitud agrego que la situación no cambia en su esencia por la introducción de la redundancia o la corrección de errores.) En el mundo discreto de la computación, no hay métrica significativa en la cual "pequeños" cambios y "pequeños" efectos vayan de la mano, y nunca los habrá."

Jerarquías conceptuales inéditas y dispositivos digitales a gran escala están en la base de la nueva oleada de la última revolución de las TICs. Volveremos sobre este asunto. Por lo pronto, hace 30 años la comunidad académica estaba en grandes dificultades para pensar de qué se trataba la computación, y más aún cómo enseñar a programar los nuevos dispositivos. Ahora también. Así, se inventaron numerosas disciplinas que intentaron pensar y enseñar las cosas nuevas con las antiguas ideas (finalmente, es lo que ocurre en esos momentos históricos especiales). Se inventó la "ingeniería del software" y se crearon sub-disciplinas como el "mantenimiento" de los programas o sistemas…viejas ideas de las ingenierías mecánicas y de construcciones, aplicadas en vano a la nueva criatura que nada tenía que ver con ellas.

Finalmente, lo único que las computadoras pueden hacer por nosotros es manipular símbolos y producir resultados de dichas manipulaciones; solo que la cantidad de símbolos así como las

manipulaciones realizadas son varios órdenes de magnitud mayores que las que podemos concebir: desconciertan completamente nuestra imaginación y por lo tanto no debemos intentar imaginarlos. Pero antes de que una computadora pueda realizar alguna manipulación con sentido debemos escribir un programa. ¿Pero qué cosa es un programa? Simplemente, un programa es un manipulador de símbolos abstracto, que puede convertirse en uno concreto suministrándole una computadora. ¿Y qué aspecto tienen estos manipuladores simbólicos abstractos? Simplemente son fórmulas más o menos elaboradas de un sistema formal. Así las cosas, la tarea del programador consiste en derivar estas fórmulas, y conocemos una única manera razonable de realizarla: mediante la manipulación de símbolos del sistema. Construimos los manipuladores de símbolos abstractos mediante la manipulación de símbolos humana.

En consecuencia, las ciencias de la computación están en la exacta intersección entre la manipulación de símbolos mecanizada y humana, habitualmente llamadas “computación” y “programación”, respectivamente. Esto nos da una pista muy precisa sobre la ubicación de la computación en el contexto de las disciplinas científicas: en la dirección de la matemática y la lógica aplicada, pero trascendiendo largamente las fronteras de lo que hoy conocemos pues las ciencias de la computación se interesan por la utilización de los métodos formales en una escala muchísimo mayor de lo que hemos sido testigos hasta el momento.

Finaliza Dijkstra:

“En el largo plazo espero que las ciencias de la computación trasciendan a sus disciplinas madres, matemática y lógica, efectivamente realizando una parte significativa del Sueño de Leibniz de proveer un cálculo simbólico como una alternativa al razonamiento humano. (Por favor, note la diferencia entre "imitar" y "proveer una alternativa a": a las alternativas se les permite ser mejores.)”

En efecto, entregar un libro sobre lógica para informática a los estudiantes es darles una herramienta no solamente útil sino imprescindible para poder aprender realmente a pensar, construir y reflexionar sobre los programas. Veremos que mucho más.

Unos 25 años después de las brillantes reflexiones anteriores, se produjo una nueva disrupción tecnológica. En un informe para el Ministerio de Ciencia y Tecnología del año 2008 - publicado como el *Libro Blanco de las TICs* - varios académicos, empresarios y profesionales señalamos que nos enfrentábamos a una oleada de novedosos dispositivos, pequeños, poderosos y portátiles al servicio de las personas. En efecto, la vida de las personas, la industria, los servicios se vieron invadidos por teléfonos, tabletas, sensores y otros tipos de diminutos aparatos que han ido ganando espacios impensados hasta entonces. Muy poco después, en buena parte por la invasión de estos dispositivos, se produjo la segunda ola de esta nueva revolución tecnológica, conocida popularmente como Big Data e Internet de las Cosas. Datos en una escala nunca antes conocida se producen, transmiten, guardan y se procesan, y se generan nuevos conocimientos…

En verdad, ninguna novedad radical se había producido en cuanto a las capacidades de generar, transmitir y guardar grandes cantidades de datos; lo efectivamente nuevo es la habilidad de transformar datos masivos en conocimientos, ese proceso se denomina inferencia. En torno de la inferencia es que se han hecho populares algunos nombres como Data Science,

Big Data Analytics, Deep Learning, Machine Learning, entre otros, que posiblemente sean los nombres de las disciplinas y las profesiones que desempeñarán muchos de los actuales estudiantes de informática, matemática, economía, sociología.

Nuevamente cabe preguntarse cuál debe ser la base de la educación de estas nuevas generaciones de estudiantes. Por cierto, no hay forma siquiera de comprender de qué tratan estas disciplinas sin conocimientos profundos de lógica y matemática, aplicadas a una escala nunca antes conocida. Tal parece que estamos más cerca de presenciar el nacimiento de un cálculo simbólico alternativo al razonamiento humano, el Sueño de Leibniz.

Lógica para Informática resulta entonces no solamente un texto sobre los fundamentos básicos de las profesiones tradicionales de las ciencias de la computación, también es parte esencial de la formación de las futuras profesiones que asoman en su horizonte.

Recorrer los capítulos es una encantadora excursión a través de los formalismos de la lógica, sus alcances como herramienta de razonamiento y sus aplicaciones, particularmente a la computación y la programación. Comenzando con razonamientos simples y la lógica proposicional, el primer capítulo trata a este formalismo de manera sencilla, completa y profunda, incluyendo sintaxis, semántica, cálculos y decidibilidad. El segundo capítulo presenta la lógica de predicados de primer orden de manera comprensible y progresiva, describe los resultados relevantes en relación con la sensatez y completitud de estos sistemas, incluyendo el famoso Teorema de Gödel y la artimética de Peano.

Establecidas las bases fundamentales, en el tercer capítulo se estudia la lógica modal, esto es las herramientas para razonar acerca de lo posible y lo necesario. Luego de una detallada presentación de la sintaxis y semántica, sistemas de deducción y teoría de modelos, se presentan la lógica deóntica y los sistemas multiagente, formalismos relevantes con aplicaciones no solamente a la computación sino también a la teoría del derecho y las ciencias sociales.

Finalmente, sobre la base de las lógicas clásicas y modal, el cuarto y último capítulo aborda de manera extensiva la verificación axiomática de programas, incluyendo programas secuenciales (determinísticos y no determinísticos), concurrentes y reactivos. Rigurosa y a la vez amena, la presentación va construyendo los sistemas de verificación sucesivamente, permitiendo la comprensión de cada nivel de dificultad.

El legado de Dijkstra y otros pioneros de la mejor tradición de las ciencias de la computación queda entonces a buen resguardo. *Lógica para Informática* se suma así a la biblioteca de los buenos textos para la mejor educación de los actuales futuros profesionales de la informática y de varias otras disciplinas relevantes para la construcción de una sociedad basada en el conocimiento.

Gabriel Baum

La Plata, octubre de 2016

Introducción

Contenido del libro

El contenido de *Lógica para Informática* se basa en la asignatura Lógica e Inteligencia Artificial (capítulos 1, 2 y 3, sobre la lógica proposicional o de enunciados, la lógica de predicados de primer orden y la lógica modal, respectivamente) y en parte de las asignaturas Teoría de la Computación y Verificación de Programas y Teoría de la Computación y Verificación de Programas Avanzada (capítulo 4, sobre la verificación axiomática de programas), asignaturas que los autores dictamos desde hace tiempo en la Licenciatura en Informática de la Universidad Nacional de La Plata. Los cuatro tópicos referidos conforman el objeto de estudio de este libro: la lógica (matemática), considerando tres de las lógicas más difundidas, y una de sus aplicaciones más interesantes en el contexto de la informática.

El título del libro puede parecer restrictivo, pero nuestra intención es la contraria. El libro es efectivamente *para Informática* porque nuestra idea primaria es aportar material bibliográfico para el dictado de las asignaturas mencionadas o similares en el marco de los planes de estudio para Informática. Lo es también porque estamos convencidos de la importancia del estudio de la lógica en la formación de los profesionales de la computación. Sin embargo el libro está orientado a *informáticos y no informáticos*, nuestro propósito es *inclusivo*, por el hecho de que la lógica forma parte de numerosos planes de estudio no solo de informática y matemática, y además porque la problemática del desarrollo de programas de computadora correctos es un tema de interés cada vez más amplio.

Uno de los autores elaboró previamente dos libros dedicados parcialmente a la verificación de programas, como soporte a las dos asignaturas de verificación antes mencionadas. El contenido del cuarto capítulo de *Lógica para Informática* debe tomarse como introductorio a ellos (salvo en lo que hace a la lógica temporal, una de las instanciaciones de la lógica modal, en la que se profundiza), y además tiene como diferencial que se centra más en los aspectos lógicos. La razón principal para incluirlo es que actúe como puente entre los tipos de lógica estudiados y la disciplina de la construcción de programas correctos.

Cada capítulo fue escrito por un autor, con lo que es posible detectar algunas diferencias de estilo en la redacción de las distintas partes del libro (también en algunos casos variedad en la simbología y formato), pero hemos procurado asegurar una estructura básica común que creemos haberla logrado. El desarrollo de los temas combina rigor matemático con informalidad, con el propósito de introducir los conceptos básicos de la mejor manera posible

desde el punto de vista didáctico. Se presentan ejemplos que acompañan las definiciones y los teoremas. Asumimos que el lector tiene cierta madurez matemática, y no excluyente pero sí deseable, conocimientos sobre algorítmica y lenguajes de programación.

Acerca de la lógica y la verificación axiomática de programas

Así como la astronomía estudia los cuerpos celestes del universo y la biología la vida, el objeto de estudio de la lógica lo constituyen las formas de razonamiento. El razonamiento es el proceso por el cual se derivan conclusiones a partir de premisas, apoyándose en verdades supuestas. El razonamiento también sirve para justificar ciertas verdades, es decir para determinar si una verdad es consecuencia (formal) de conocimientos previamente aceptados.

Un razonamiento es correcto si la manera en que está construido garantiza la conservación de la verdad: si las premisas son verdaderas entonces la conclusión es necesariamente verdadera. Y es incorrecto si su construcción es defectuosa, si no hay garantía acerca del valor de verdad de la conclusión. La lógica investiga los principios por los cuales algunos razonamientos son correctos y otros no. Cuando un razonamiento es correcto, lo es por su estructura lógica y no por el contenido específico del argumento o el lenguaje utilizado. Por esta razón la lógica se considera una ciencia formal, como la matemática, en vez de una ciencia empírica. A un razonamiento correcto se lo denomina deducción.

Por ejemplo, consideremos el siguiente razonamiento: (a) Los griegos son hombres. (b) Los hombres son mortales. (c) Por lo tanto los griegos son mortales. Este razonamiento posee la siguiente estructura lógica: (a) A es B. (b) B es C. (c) Por lo tanto A es C. La estructura refleja una forma de razonamiento correcto, conocida como silogismo.

En un principio la lógica no tuvo el sentido de estructura formal estricta que tiene ahora. El tratamiento sofisticado de la lógica moderna aparentemente proviene de la tradición griega. Aristóteles fue el primero en formalizar los razonamientos, utilizando letras para representar términos. Sostuvo que la verdad se manifiesta en el juicio verdadero y el argumento válido en el silogismo. En su principal obra lógica desarrolló el silogismo (sistema lógico de estructura rígida), formalizó el cuadro de oposición de los juicios, categorizó las formas válidas del silogismo, y reconoció y estudió los argumentos inductivos, base de la ciencia experimental cuya lógica está estrechamente ligada al método científico.

Filósofos racionalistas como R. Descartes, B. Pascal y G. Leibniz aportaron a través del desarrollo de la matemática temas que van a marcar una importante evolución en la lógica. De especial importancia fueron las ideas de Descartes y Leibniz sobre una ciencia y lenguaje universales, especificados con precisión matemática, sobre la base de que la sintaxis de las palabras debería estar en correspondencia con las entidades designadas como individuos o elementos metafísicos, lo que haría posible un cálculo o computación mediante algoritmos en el descubrimiento de la verdad. Aparecieron así los primeros intentos y realizaciones de

máquinas de cálculo (Pascal y Leibniz), y aunque su desarrollo no fue eficaz, sin embargo constituyeron el antecedente inmediato del crecimiento de la lógica a partir del siglo XX.

A partir de la segunda mitad del siglo XIX la lógica se rebeló. G. Boole publicó en 1847 *El análisis matemático de la lógica* y en 1854 *Las leyes del pensamiento*. Boole construyó un cálculo en el que los valores de verdad se representan mediante el 0 (falsedad) y el 1 (verdad), a los que se les aplican operaciones matemáticas como la suma y la multiplicación. En el mismo año 1847, A. De Morgan publicó *Lógica formal*, donde introdujo las famosas leyes que llevan su nombre e intentó generalizar la noción de silogismo. Otro aporte importante lo hizo J. Venn, quien con *Lógica simbólica* introdujo los famosos diagramas que llevan su nombre.

Pero la más importante revolución de la lógica vino de la mano de Frege, frecuentemente considerado el lógico más importante de la historia junto a Aristóteles. Con su trabajo de 1879 *La conceptografía*, presentó por primera vez un sistema completo de lógica de predicados. También desarrolló la idea de un lenguaje formal y definió la noción de prueba. Estas ideas constituyeron una base teórica fundamental para el desarrollo de las computadoras y las ciencias de la computación. Los contemporáneos de Frege pasaron por alto sus contribuciones, probablemente a causa de la complicada notación que utilizó. En 1893 y 1903, Frege publicó en dos volúmenes *Las leyes de la aritmética*, donde intentó deducir toda la matemática a partir de la lógica, lo que se conoció como el proyecto logicista. Pero su sistema contenía una contradicción: la paradoja de Russell.

El siglo XX sería el siglo de los enormes desarrollos en lógica. La lógica pasó a estudiarse por su interés intrínseco y no solo por sus virtudes como propedéutica. En 1910 B. Russell y A. Whitehead publicaron *Principia mathematica*, un trabajo monumental en el que lograron plasmar gran parte de la matemática a partir de la lógica, evitando caer en paradojas. Llegarían luego noticias (solo en algún aspecto) devastadoras, en el sentido de que la lógica no era la panacea universal que algunos pretendían, quizá un poco ingenuamente. Por un lado K. Gödel probaba en 1931 la imposibilidad de llevar a cabo el programa de Hilbert, consistente en formalizar completamente la matemática clásica reemplazando sus conceptos por cadenas de símbolos y el razonamiento por una mera manipulación de dichas cadenas en base a reglas mecánicas. Y por el otro, A. Church y A. Turing resolvían en 1936 de manera negativa el problema de decisión: no es decidible determinar si una fórmula es un teorema de un sistema axiomático de primer orden.

Además de la lógica proposicional y de predicados (lógica clásica), en el siglo XX se desarrollaron otros sistemas lógicos, entre ellos diversas lógicas modales. Los sistemas lógicos clásicos son los más estudiados y utilizados, se caracterizan por incorporar principios tradicionales que otras lógicas rechazan, como el principio del tercero excluido, el principio de no contradicción, el principio de explosión y la monotonicidad de la implicación. Las lógicas no clásicas rechazan uno o más principios de la lógica clásica. Por ejemplo, la lógica difusa rechaza el principio del tercero excluido y propone un número infinito de valores de verdad. Las lógicas modales son lógicas que están diseñadas para tratar con expresiones que califican la verdad de los juicios. Así por ejemplo la expresión “siempre” califica a un

juicio verdadero como verdadero en todo momento (no es lo mismo decir que está lloviendo a decir que siempre está lloviendo).

Nuestro libro dedica sus primeros tres capítulos al estudio de lógicas clásicas y no clásicas: la lógica proposicional y de predicados de la primera familia, y la lógica modal de la segunda. En el cuarto y último capítulo se estudia el uso de dichas lógicas (en el caso de la lógica modal, una de sus instanciaciones más difundidas que es la lógica temporal) para la verificación formal de programas, y como efecto colateral pero no menos importante para el desarrollo sistemático de software, práctica iniciada a fines de los años 1960 por los impulsores de la programación estructurada, como R. Floyd, C. Hoare, E. Dijkstra y D. Gries. Se trató en un sentido de poner en pie de igualdad a los programas de computadora con sus especificaciones, descriptas formalmente mediante lenguajes matemáticos. De esta manera los objetos de la disciplina de programación en sus distintas variantes (síntesis de programas, verificación de programas, análisis de programas, etc.) pasaron a ser fórmulas con programas y especificaciones, manipulables con mecanismos formales como los sistemas axiomáticos que vamos a describir en este libro.

La contribución que se considera fundacional para la verificación formal de programas es la de Floyd en 1967, en la que mediante métodos axiomáticos se trata la prueba de correctitud de programas empleando aserciones inductivas sobre diagramas de flujo. El primer sistema de prueba sobre programas estructurados lo presentó Hoare en 1969; este trabajo impulsó sobremanera la disciplina completa de la verificación de programas. El desarrollo sistemático de programas secuenciales tomando como guía los métodos axiomáticos se inició en 1976 con Dijkstra. En cuanto a la lógica temporal, su empleo en la verificación de programas partió del trabajo de A. Pnueli en 1977. En este contexto, entre 1981 y 1982 J. Queille, J. Sifakis, E. Emerson y E. Clarke comenzaron el desarrollo de herramientas para chequear automáticamente que los programas satisfagan especificaciones escritas con aserciones de la lógica temporal, aproximación que se conoce como *model checking*. Cabe también destacar el trabajo de 1988 de K. Chandy y J. Misra, que sistematiza la síntesis de programas concurrentes a partir de especificaciones expresadas en lógica temporal; se lo considera fundacional como lo fue el de Dijkstra para la programación secuencial.

Cómo leer este libro

Cada capítulo tiene al final ejercicios para resolver y referencias bibliográficas. Dejamos para el último capítulo mayoritariamente ejercicios de demostraciones formales con axiomas y reglas, haciendo hincapié en los primeros en los fundamentos de la lógica clásica y modal. La mayor extensión del capítulo 4 con respecto al resto debe entenderse por ser en realidad dos capítulos en uno, la primera parte dedicada a la lógica clásica y la segunda a la lógica modal.

La forma natural de leer este libro es recorriendo secuencialmente sus cuatro capítulos, pero no es la única.

Una manera alternativa de lectura que sugerimos, para ir de lo más elemental a lo más complejo, es: (a) Los capítulos 1 y 2 completos, de lógica proposicional y de lógica de predicados de primer orden (lógica clásica). (b) La sección de conceptos básicos del capítulo 3, de lógica modal. (c) El capítulo 4, de lógica de programas (así lo denominamos para enfatizar la relación estudiada de la verificación de programas con los sistemas axiomáticos), sin considerar la sección final de notas. (d) Las secciones de lógica deóntica y sistemas multiagente del capítulo 3. Y (e) las notas finales del capítulo 4.

También se puede considerar una tercera posibilidad de lectura del libro: (a) Los capítulos 1 y 2 completos. (b) La primera parte del capítulo 4, dedicada a la lógica de programas de entrada/salida, porque se basa en la lógica de predicados de primer orden. (c) La sección de conceptos básicos del capítulo 3. (d) La segunda parte del capítulo 4, dedicada a la lógica de programas reactivos, porque se basa en la lógica temporal. (e) Las secciones de lógica deóntica y sistemas multiagente del capítulo 3. Y (f) las notas finales del capítulo 4.

La sección de sistemas de primer orden del capítulo 2, que trata sobre las extensiones de la lógica de primer orden con axiomas propios del dominio considerado, sirve de base para los métodos de verificación del capítulo 4, específicamente teniendo en cuenta el dominio de los números enteros, en términos de los cuales se desarrollan los métodos. Las secciones de lógica deóntica y sistemas multiagente del capítulo 3 ejemplifican lo tratado en la sección previa de conceptos básicos, considerando su aplicación en la ética y el derecho y en la interacción entre distintos agentes cognitivos, respectivamente (la lógica temporal se describe directamente en el capítulo 4). Algunas notas finales del capítulo 4 en realidad aplican a todos (por ejemplo las referencias a la completitud, la decidibilidad y la composicionalidad).

Es nuestro deseo que este libro sea una guía de estudio para los alumnos, un apoyo para los docentes que dictan o quieran dictar estos contenidos o contenidos afines, un aporte para la formación básica de los alumnos de la UNLP y de otras universidades, y un estímulo para profundizar en la investigación de los tópicos tratados.

Claudia Pons, Ricardo Rosenfeld, Clara Smith

La Plata, octubre de 2016

CAPÍTULO 1
Lógica proposicional

Claudia Pons

Introducción

La lógica proposicional, también conocida como lógica de enunciados, es un sistema formal cuyos elementos representan *proposiciones* o *enunciados*. Esta lógica no tiene, por sí misma, mucha utilidad para la representación del conocimiento. Está justificado detenerse en ella porque permite introducir de una manera sencilla algunos conceptos que, explicados directamente para la lógica de predicados (a ser introducida en el capítulo siguiente), son más difíciles de captar.

Nos interesa examinar los mecanismos de razonamiento con precisión matemática. Esta precisión requiere que el lenguaje que usemos no dé lugar a confusiones, lo cual conseguimos mediante un lenguaje simbólico donde cada símbolo tenga un significado bien definido. Dada una frase en lenguaje natural, en primer lugar, podemos observar si se trata de una frase simple o de una frase compuesta. Una frase simple consta de un sujeto y un predicado. Por ejemplo:

- Java es un lenguaje de programación.
- Android es un sistema operativo moderno.

Una frase compuesta se forma a partir de frases simples por medio de algún término de enlace (o conectiva). Por ejemplo:

- Java es un lenguaje de programación y Java es compatible con Android.
- Si Android es un sistema operativo moderno entonces Android soporta Java.

En segundo lugar, vamos a suponer que todas las frases simples pueden ser verdaderas o falsas. Ahora bien, en castellano hay frases que no son ni verdaderas ni falsas (exclamaciones, órdenes, preguntas), por tanto tenemos que usar un término diferente. Hablaremos de enunciados (o proposiciones) para referirnos a frases que son verdaderas o falsas. Y distinguiremos entre enunciados simples (atómicos) o enunciados compuestos.

Denotamos los enunciados con letras mayúsculas (A, B, C). Para construir enunciados compuestos introducimos símbolos para las conectivas. Las conectivas más comunes y los símbolos que emplearemos para denotarlas son los siguientes:

- $\neg A$ Negación de A
- $A \wedge B$ Conjunción de A y B
- $A \vee B$ Disyunción de A o B
- $A \rightarrow B$ Si A entonces B
- $A \leftrightarrow B$ A si y solo si B

Así, los enunciados compuestos vistos antes pueden escribirse simbólicamente de la siguiente forma:

- $A \wedge B$
- $C \rightarrow D$

A simboliza "Java es un lenguaje de programación", B simboliza "Java es compatible con Android", C simboliza "Android es un sistema operativo moderno" y D simboliza "Android soporta Java".

Nótese que cuando un enunciado se traduce al lenguaje simbólico, lo que queda es su "estructura lógica", que puede ser común a varios enunciados diferentes. Esto nos permite analizar las formas de razonamiento, ya que un razonamiento tiene que ver con la estructura lógica de los enunciados de la argumentación y no con su significado. Observemos los siguientes casos:

Las premisas son verdaderas y la conclusión también es verdadera

Consideremos el siguiente razonamiento:

Si Juan es mendocino entonces Juan es argentino.
Si Juan es argentino entonces Juan es sudamericano.
Por lo tanto: si Juan es mendocino entonces Juan es sudamericano.

Este razonamiento posee la siguiente estructura lógica:

Si A entonces B.
Si B entonces C.
Por lo tanto: si A entonces C.

Esta estructura refleja una forma de razonamiento correcto, conocida como silogismo. Consideremos ahora el siguiente razonamiento:

Si Juan es mendocino entonces Juan es sudamericano.
Si Juan es argentino entonces Juan es sudamericano.
Por lo tanto: si Juan es mendocino entonces Juan es argentino.

Este razonamiento posee la siguiente estructura de razonamiento:

Si A entonces B.
Si C entonces B.
Por lo tanto: si A entonces C.

Esta estructura evidencia una forma incorrecta de razonar, que en este caso permitió obtener una conclusión verdadera a partir de premisas verdaderas. Sin embargo la misma estructura de razonamiento podría instanciarse con otros enunciados que dejarían en evidencia su incorrección, tal como veremos a continuación.

Las premisas son verdaderas y la conclusión es falsa

El razonamiento correcto preserva la verdad, no es posible partir de premisas verdaderas y llegar a conclusiones falsas a través de un razonamiento correcto. Esta situación puede ocurrir únicamente si aplicamos un razonamiento incorrecto, como en el siguiente ejemplo:

Si Juan es mendocino entonces Juan es argentino.
Si Juan es salteño entonces Juan es argentino.
Por lo tanto: si Juan es mendocino entonces Juan es salteño.

Esta es otra instancia de la estructura de razonamiento incorrecto vista antes:

Si A entonces B.
Si C entonces B.
Por lo tanto: si A entonces C.

Con este ejemplo la falla en la estructura de razonamiento queda en evidencia. ¿Qué ocurre con un razonamiento que parte de premisas falsas? Veamos los casos:

Algunas premisas son falsas y la conclusión también es falsa

Consideremos las mismas estructuras de razonamiento de antes, pero ahora utilizando premisas falsas:

Si Juan es argentino entonces Juan es africano.
Si Juan es africano entonces Juan es asiático.
Por lo tanto: si Juan es argentino entonces Juan es asiático.

Observamos que se trata de un razonamiento correcto (nuevamente el esquema del silogismo), que nos ha permitido deducir información falsa a partir de premisas falsas. Utilicemos ahora el razonamiento incorrecto anterior:

Si Juan es chino entonces Juan es sudamericano.
Si Juan es peruano entonces Juan es sudamericano.
Por lo tanto: si Juan es chino entonces Juan es peruano.

También esta forma de razonamiento incorrecto nos ha permitido deducir información falsa a partir de premisas falsas.

Algunas premisas son falsas y la conclusión es verdadera

Es posible partir de premisas falsas y arribar a conclusiones verdaderas. Podríamos decir que se llega a la verdad "por casualidad". Veamos el siguiente ejemplo que nuevamente aplica el silogismo como esquema de razonamiento correcto:

Si Juan es argentino entonces Juan es africano.
Si Juan es africano entonces Juan es sudamericano.
Por lo tanto: si Juan es argentino entonces Juan es sudamericano.

Una situación similar ocurre si aplicamos una forma incorrecta de razonar, como en el siguiente ejemplo:

Si Juan es mendocino entonces Juan es africano.
Si Juan es argentino entonces Juan es africano.
Por lo tanto: si Juan es mendocino entonces Juan es argentino.

En resumen, un razonamiento es directamente incorrecto cuando a partir de premisas verdaderas permite arribar a una conclusión falsa, o bien es incorrecto porque tiene la estructura de un razonamiento incorrecto (aunque la conclusión sea verdadera). La corrección de la forma solamente garantiza que si las premisas son verdaderas entonces lo será también la conclusión. Este caso es de gran importancia en el método científico, ya que permite razonar correctamente, pero sobre hipótesis que podrían ser falsas. La verdad de la conclusión no nos asegura nada acerca de la verdad de las premisas.

Sintaxis: el lenguaje simbólico de la lógica

Para estudiar los principios del razonamiento, la lógica necesita mediante sistemas formales en primer término capturar y formalizar las estructuras del lenguaje natural en un lenguaje

simbólico, para luego formalizar los mecanismos de razonamiento que se aplican sobre dichas estructuras lingüísticas.

El lenguaje simbólico consta de un conjunto de símbolos primitivos (el alfabeto o vocabulario) y un conjunto de reglas de formación (la gramática) que nos dice cómo construir fórmulas bien formadas a partir de los símbolos primitivos.

Alfabeto

El alfabeto de un sistema formal es el conjunto de símbolos que pertenecen al lenguaje del sistema. Si L es el nombre del sistema de lógica proposicional, entonces el alfabeto de L consiste en:

- Una cantidad finita pero arbitrariamente grande de variables proposicionales (o variables de enunciado). En general se las toma del alfabeto latino, empezando por la letra p, luego q, r, etc., y utilizando subíndices cuando es necesario. Las variables de enunciado representan enunciados simples como "está lloviendo" o "los metales se expanden con el calor".
- Un conjunto de operadores lógicos o conectivas: $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$.
- Dos signos de puntuación: el paréntesis izquierdo y el paréntesis derecho. Su única función es desambiguar ciertas expresiones, como veremos (tal como se hace con las operaciones aritméticas por ejemplo: sin una convención definida, la expresión $2 + 2 \div 2$ puede significar tanto $(2 + 2) \div 2$ como $2 + (2 \div 2)$).

Gramática

Una vez definido el alfabeto, el siguiente paso es determinar qué combinaciones de símbolos pertenecen al lenguaje del sistema. Esto se logra mediante una gramática formal. La misma consiste en un conjunto de reglas que definen recursivamente las cadenas de caracteres que pertenecen al lenguaje. A las cadenas de caracteres construidas según estas reglas se las llama *fórmulas bien formadas*, y también se las conoce como *formas enunciativas*. Las reglas del sistema L son (para simplificar la notación también recurrimos a las letras A, B, C, …, para denotar las formas enunciativas, siempre que quede claro por contexto su uso):

i. Las variables de enunciado del alfabeto de L son formas enunciativas.
ii. Si A y B son formas enunciativas de L, entonces también lo son $(\neg A)$, $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$ y $(A \leftrightarrow B)$.
iii. Solo las expresiones que pueden ser generadas mediante las cláusulas i y ii en un número finito de pasos son formas enunciativas de L.

Según estas reglas, las siguientes cadenas de caracteres son ejemplos de formas enunciativas:

- p, q, r, por la definición (i).
- $(\neg p)$, $(q \wedge r)$, $(p \rightarrow q)$, por la definición (ii) y la línea anterior.
- $(((\neg p) \vee (q \wedge r)) \rightarrow (p \rightarrow q))$, por la definición (ii) y la línea anterior.

Y los siguientes son ejemplos de fórmulas que no son formas enunciativas:

Fórmula	Error	Corrección
(p)	Sobran paréntesis	p
$\neg$ (p)	Mal los paréntesis	$(\neg p)$
$p \rightarrow q$	Faltan paréntesis	$(p \rightarrow q)$

Dado que la única función de los paréntesis es desambiguar las fórmulas, en general se acostumbra omitir los paréntesis externos, ya que no cumplen ninguna función. Así por ejemplo, fórmulas como $p \wedge q$, $(\neg p) \rightarrow q$, etc., se consideran bien formadas. Otra convención habitual para abreviar paréntesis es que las conjunciones y las disyunciones tienen "menor jerarquía" que los condicionales y bicondicionales. Lo mismo sucede con la negación respecto de la conjunción y disyunción. Esto significa que dada una fórmula sin paréntesis, las conjunciones y las disyunciones deben resolverse antes que los condicionales y los bicondicionales, y las negaciones antes que las conjunciones y disyunciones. Por ejemplo:

Fórmula	Lectura correcta	Lectura incorrecta
$p \wedge q \rightarrow r$	$(p \wedge q) \rightarrow r$	$p \wedge (q \rightarrow r)$
$\neg p \leftrightarrow q \vee r$	$(\neg p) \leftrightarrow (q \vee r)$	$((\neg p) \leftrightarrow q) \vee r$

Estas convenciones son análogas a las que existen en el álgebra elemental, ya referida recién, donde por ejemplo la multiplicación y la división deben resolverse antes que la suma y la resta. Sin una convención, la ecuación $2 + 2 \times 2$ podría interpretarse como $(2 + 2) \times 2$ o como $2 + (2 \times 2)$. En el primer caso el resultado es 8 y en el segundo caso 6. Como la multiplicación debe resolverse antes que la suma, el resultado correcto en este caso es 6.

Denotaremos con letras minúsculas p, q, r, ..., a las variables que designan enunciados simples arbitrarios. Nótese la distinción entre los usos de las letras p, q, r, …, y las letras A, B, C, ... Las primeras son variables que pueden ser sustituidas por enunciados simples particulares. Las últimas son meras etiquetas que designan enunciados en general. Estos componentes nos permitirán describir las propiedades que poseen los enunciados y las conectivas.

Semántica: interpretación y satisfacción

Como todo enunciado simple es verdadero o falso, una variable de enunciado tomará uno u otro valor de verdad: V (verdadero) o F (falso). La verdad o falsedad de un enunciado compuesto depende de la verdad o falsedad de los enunciados simples que lo constituyen, y de la forma en que están conectados.

Primeramente vamos a analizar el significado de cada una de las conectivas, mediante *tablas de verdad*.

Negación

Sea A un enunciado, siendo irrelevante su significado. Denotaremos con $\neg$ A a su negación. Si A es verdadero entonces $\neg$ A es falso, y recíprocamente si A es falso entonces $\neg$ A es verdadero. La siguiente es la tabla de verdad que especifica el significado de esta conectiva:

A	$\neg$ A
V	F
F	V

La conectiva $\neg$ da lugar a una función de verdad llamada $f^{\neg}$ que tiene como dominio y codominio al conjunto {V, F} y se define así:

- $f^{\neg}$(V) = F
- $f^{\neg}$(F) = V

Conjunción

Sean A y B dos enunciados, denotamos con A $\wedge$ B a la conjunción de ámbos. Su tabla de verdad es la siguiente:

A	B	A $\wedge$ B
V	V	V
V	F	F
F	V	F
F	F	F

La conectiva $\wedge$ define una función de verdad $f^{\wedge}$ de dos argumentos:

- $f^{\wedge}$ (V, V) = V
- $f^{\wedge}$ (V, F) = F
- $f^{\wedge}$ (F, V) = F
- $f^{\wedge}$ (F, F) = F

Disyunción

Sean A y B dos enunciados. En castellano tenemos dos usos distintos para la disyunción "o". Elegimos "A o B o ámbos", que denotamos con $A \vee B$. Su tabla de verdad es:

A	B	$A \vee B$
V	V	V
V	F	V
F	V	V
F	F	F

La conectiva $\vee$ define una función de verdad $f^{\vee}$ de dos argumentos, como la anterior:

- $f^{\vee}$ (V, V) = V
- $f^{\vee}$ (V, F) = V
- $f^{\vee}$ (F, V) = V
- $f^{\vee}$ (F, F) = F

Nótese que el otro uso de la disyunción, es decir "A o B pero no ámbos", se puede simbolizar mediante $(A \vee B) \wedge \neg (A \wedge B)$.

Condicional

Sean A y B dos enunciados. Utilizaremos $A \rightarrow B$ para representar el enunciado "A implica a B" o "si A entonces B". En este caso el significado intuitivo de esta frase genera algunos conflictos con su significado formal. La tabla de verdad de esta conectiva es la siguiente:

A	B	$A \rightarrow B$
V	V	V
V	F	F
F	V	V
F	F	V

De la misma forma que las anteriores, la conectiva $\rightarrow$ define una función de verdad de dos argumentos:

- $f^{\rightarrow}$ (V, V) = V
- $f^{\rightarrow}$ (V, F) = F
- $f^{\rightarrow}$ (F, V) = V
- $f^{\rightarrow}$ (F, F) = V

La dificultad radica en el valor de verdad V asignado a (A → B) en los casos en que A es falso. Intuitivamente podemos considerar a la implicación como un contrato: cuando su antecedente no se cumple su consecuente puede tanto cumplirse como no cumplirse, y en ambos casos el contrato no se ha quebrado y podemos considerarlo verdadero.

Por ejemplo, consideremos el enunciado “Si Alexia limpia su cuarto entonces su madre le da dinero”. Se simboliza con (A → B), siendo A el enunciado “Alexia limpia su cuarto” y B el enunciado “su madre le da dinero”. Veámoslo como un contrato entre Alexia y su madre:

- ¿Qué ocurre si Alexia limpia su cuarto? Su madre debe darle dinero pues ése es el contrato (se corresponde con el caso $f^{\rightarrow}$(V, V) = V). En caso contrario el contrato se habría quebrado pues la madre de Alexia habría incumplido su promesa (se corresponde con el caso $f^{\rightarrow}$(V, F) = F).
- ¿Y qué ocurre si Alexia no limpia su cuarto? Su madre podría no darle el dinero (se corresponde con el caso $f^{\rightarrow}$(F, F) = V). El contrato entre ambas se cumple. Finalmente, aunque Alexia no limpie su cuarto su madre podría darle igualmente el dinero (se corresponde con el caso $f^{\rightarrow}$(F, V) = V). El contrato no se ha quebrado, pues no prohíbe la entrega del dinero. Nótese que la situación sería diferente si el contrato expresara: “Su madre le da dinero si y solo si Alexia limpia su cuarto”.

Otro ejemplo interesante es el siguiente enunciado matemático: si $n \geq 2$ entonces $n^2 \geq 4$, el cual es verdadero independientemente del valor que tome n. El punto a recordar es que la única circunstancia en la que el enunciado A → B es falso se da cuando A es verdadero y B es falso.

Bicondicional

Sean A y B dos enunciados. Denotamos el enunciado “A si y solo si B” o “A equivale a B” con A ↔ B. Este enunciado será verdadero cuando A y B tengan el mismo valor de verdad (ámbos verdaderos o ámbos falsos), y solo en dicho caso. La tabla de verdad es la siguiente:

A	B	A ↔ B
V	V	V
V	F	F
F	V	F
F	F	V

Como antes, la conectiva ↔ define una función de verdad de dos argumentos:

- $f^{\leftrightarrow}$(V, V) = V
- $f^{\leftrightarrow}$(V, F) = F
- $f^{\leftrightarrow}$(F, V) = F
- $f^{\leftrightarrow}$(F, F) = V

En lo que sigue veremos que el valor de verdad de un enunciado compuesto depende de los valores de verdad de los enunciados simples que lo forman, aplicando las funciones de verdad de las conectivas.

Tablas de verdad para enunciados compuestos

La tabla de verdad de una forma enunciativa cualquiera establece, para cada asignación de valores de verdad sobre las variables de enunciado involucradas, el valor que toma, y se obtiene usando las tablas de verdad de las conectivas analizadas previamente. Dicha tabla es una representación gráfica de una función de verdad, cuyo número de argumentos es igual al número de variables distintas que intervienen. Es decir, a una forma enunciativa con n variables diferentes, siendo n > 0, le corresponde una función de verdad de n argumentos, y la tabla de verdad tendrá 2^n filas, una para cada una de las posibles combinaciones de valores de verdad. Por ejemplo:

Para el caso de $(\neg p) \vee q$ se tiene:

p	q	$(\neg p)$	$(\neg p) \vee q$
V	V	F	V
V	F	F	F
F	V	V	V
F	F	V	V

Y en el caso de $p \rightarrow (q \wedge r)$ tenemos:

p	q	r	$(q \wedge r)$	$p \rightarrow (q \wedge r)$
V	V	V	V	V
V	V	F	F	F
V	F	V	F	F
V	F	F	F	F
F	V	V	V	V
F	V	F	F	V
F	F	V	F	V
F	F	F	F	V

Notemos que existen 2^k funciones de verdad distintas de n argumentos, con $k = 2^n$, que corresponden a todas las formas posibles de disponer los valores V y F en la última columna de una tabla de verdad de k filas. Claramente con n variables de enunciado se pueden construir infinitas formas enunciativas asociadas a una misma función de verdad. Para analizar esto más a fondo necesitamos algunas definiciones.

Definición. Interpretación y satisfacción. En términos generales, una *interpretación* es una función que relaciona los elementos de los dominios sintáctico y semántico de la lógica considerada. En el caso particular de la lógica proposicional, una interpretación I consiste en una *función de valuación* v_I que asigna a cada variable de enunciado el valor de verdad V o F. Siendo P = {p_1, p_2, …} el conjunto de variables de enunciado, se escribe:

$$v_I : P \rightarrow \{V, F\}$$

Para extender el dominio de la función de valuación de las variables de enunciado a las formas enunciativas en general, basta con definir una regla semántica para cada una de las reglas sintácticas de la gramática:

- $|=_I$ p si y solo si v_I (p) = V
- $|=_I \neg$ A si y solo si no es el caso que $|=_I$ A
- $|=_I$ A $\vee$ B si y solo si o bien $|=_I$ A o bien $|=_I$ B o ámbos
- $|=_I$ A $\wedge$ B si y solo si $|=_I$ A y $|=_I$ B
- $|=_I$ A $\rightarrow$ B si y solo si no es el caso que $|=_I$ A y no $|=_I$ B
- $|=_I$ A $\leftrightarrow$ B si y solo si $|=_I$ A $\rightarrow$ B y $|=_I$ B $\rightarrow$ A

El símbolo |= se utiliza para las definiciones semánticas. Con $|=_I$ se hace referencia a la interpretación I. Se dice que una interpretación I *satisface* una forma enunciativa A si y solo si se cumple v_I (A) = V. También se puede decir que A se satisface con I, o en lenguaje más coloquial que A es verdadera con dicha interpretación. Como vimos, simbólicamente se expresa así: $|=_I$ A.

Definición. Tautología y contradicción. Una forma enunciativa es una *tautología* si siempre toma el valor de verdad V, considerando todas y cada una de las posibles asignaciones de valores de verdad a las variables de enunciado que contiene. Si en cambio siempre toma el valor de verdad F, la forma enunciativa se conoce como *contradicción*. El método para determinar si una forma enunciativa es una tautología o una contradicción consiste en construir su tabla de verdad. Por ejemplo:

- p $\vee$ ($\neg$ p) es una tautología
- p $\wedge$ ($\neg$ p) es una contradicción
- p $\vee$ q no es ni una tautología ni una contradicción

Claramente, toda tautología con n variables tiene asociada una misma función de verdad de n argumentos, es decir la misma tabla de verdad de 2^n filas donde la última columna siempre contiene el valor V. Una situación similar ocurre para las contradicciones con el valor F.

Definición. Modelo. Una interpretación I es un *modelo de una forma enunciativa A* si A se satisface con I, es decir si $\models_I$ A. Una interpretación es un *modelo de un conjunto de formas enunciativas* si es un modelo de cada una de ellas.

Implicación lógica y equivalencia lógica

Sean A y B dos enunciados. Diremos que "A implica lógicamente a B" o que "B es consecuencia lógica de A" (lo denotaremos con A ⇒ B) si la forma enunciativa A → B es una tautología. Y diremos que "A es lógicamente equivalente a B" (lo denotaremos con A ⇔ B) si la forma enunciativa A ↔ B es una tautología. Por ejemplo:

- p ∧ q implica lógicamente a p
- ¬ (p ∧ q) es lógicamente equivalente a (¬ p) ∨ (¬ q)
- ¬ (p ∨ q) es lógicamente equivalente a (¬ p) ∧ (¬ q)

Demostramos a continuación la última de las equivalencias lógicas de arriba. Para ello vamos a construir la tabla de verdad correspondiente a la siguiente forma enunciativa:

¬ (p ∨ q) ↔ (¬ p) ∧ (¬ q)

¬	**(p**	∨	**q)**	↔	**(¬**	**p)**	∧	**(¬**	**q)**
F	V	V	V	**V**	F	V	F	F	V
F	V	V	F	**V**	F	V	F	V	F
F	F	V	V	**V**	V	F	F	F	V
V	F	F	F	**V**	V	F	V	V	F

Es decir, comenzamos escribiendo las 2^2 filas con las combinaciones para los valores de verdad V y F de las dos variables de enunciado p y q. Y luego vamos resolviendo los valores de verdad de cada parte de la forma enunciativa, colocando el resultado parcial bajo la conectiva correspondiente.

Notar que si A y B son dos formas enunciativas lógicamente equivalentes con las mismas variables de enunciado, entonces tienen la misma función de verdad.

Las siguientes son equivalencias lógicas muy conocidas, por resultar útiles a la hora de manipular formas enunciativas:

- Ley de Doble Negación ¬ (¬ p) ⇔ p
- Ley Conmutativa de la Conjunción p ∧ q ⇔ q ∧ p

- Ley Conmutativa de la Disyunción $p \vee q \Leftrightarrow q \vee p$
- Ley Asociativa de la Conjunción $(p \wedge q) \wedge r \Leftrightarrow p \wedge (q \wedge r)$
- Ley Asociativa de la Disyunción $(p \vee q) \vee r \Leftrightarrow p \vee (q \vee r)$
- Leyes de De Morgan $\neg (p \vee q) \Leftrightarrow (\neg p) \wedge (\neg q)$
  $\neg (p \wedge q) \Leftrightarrow (\neg p) \vee (\neg q)$
- Leyes de Distribución $p \wedge (q \vee r) \Leftrightarrow (p \wedge q) \vee (p \wedge r)$
  $p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$
- Leyes de Absorción $p \wedge p \Leftrightarrow p$
  $p \vee p \Leftrightarrow p$

Formas normales

Hemos visto que a partir de toda forma enunciativa puede construirse una tabla de verdad. Vamos a formular ahora un resultado en un sentido recíproco:

Proposición. Toda función de verdad es la función de verdad determinada por una forma enunciativa *restringida*. Llamamos forma enunciativa restringida a una forma enunciativa en la que solamente figuran las conectivas $\neg$, $\wedge$, $\vee$.

Corolario. Toda forma enunciativa, que no es una contradicción, es lógicamente equivalente a una forma enunciativa restringida de la forma:

$$\bigvee_{i=1\ldots m} \bigwedge_{j=1\ldots n} Q_{ij}, \quad \text{es decir:} \quad (Q_{11} \wedge \ldots \wedge Q_{1n}) \vee \ldots \vee (Q_{m1}, \wedge \ldots \wedge Q_{mn})$$

donde cada Q_{ij} es una variable de enunciado o la negación de una variable de enunciado. Esta forma se denomina *forma normal disyuntiva.*

Idea de demostración del corolario. Dos formas enunciativas son lógicamente equivalentes si y solo si corresponden a la misma función de verdad. Dada una forma enunciativa A, obtenemos su tabla de verdad y la función de verdad que ésta define. Aplicando el método en que se basa la demostración de la proposición anterior se puede obtener una forma enunciativa en la forma deseada, correspondiente a dicha función de verdad.

Otro corolario. Toda forma enunciativa, que no es una tautología, es lógicamente equivalente a una forma enunciativa restringida de la forma:

$$\bigwedge_{i=1\ldots m} \bigvee_{j=1\ldots n} Q_{ij}, \quad \text{es decir:} \quad (Q_{11} \vee \ldots \vee Q_{1n}) \wedge \ldots \wedge (Q_{m1}, \vee \ldots \vee Q_{mn})$$

donde cada Q i j es una variable de enunciado o la negación de una variable de enunciado. Esta forma se denomina *forma normal conjuntiva.* La demostración de este corolario se basa en el anterior, las leyes de De Morgan y el hecho de que la negación de una forma enunciativa que no es una tautología no es una contradicción.

Conjuntos adecuados de conectivas

Un conjunto adecuado de conectivas es un conjunto tal que toda función de verdad puede representarse por medio de una forma enunciativa en la que solo aparezcan conectivas de dicho conjunto.

Proposición. Los pares {¬, ∧}, {¬, ∨} y {¬, →} son conjuntos adecuados de conectivas.

Los anteriores son los únicos conjuntos adecuados de conectivas con dos elementos. ¿Existen conjuntos unitarios adecuados de conectivas, es decir con una sola conectiva? Las cinco conectivas ¬, ∨, ∧, →, ↔ que hemos estudiado no constituyen por sí solas un conjunto adecuado. Pero no son las únicas conectivas posibles. De hecho cada tabla de verdad define una nueva conectiva pero con significado intuitivo no muy claro. Se debe a H. Sheffer la introducción de dos nuevas conectivas:

- **Nor.** Se denota con ↓ y no es más que la negación de la disyunción, es decir ¬ (p ∨ q). Su tabla de verdad es por lo tanto la siguiente:

p	q	p ↓ q
V	V	F
V	F	F
F	V	F
F	F	V

- **Nand.** Se denota con **|** y no es más que la negación de la conjunción, es decir ¬ (p ∧ q). Su tabla de verdad es por lo tanto:

p	q	p | q
V	V	F
V	F	V
F	V	V
F	F	V

El interés por estas conectivas se aclara en la siguiente proposición. También cabe remarcar que se aplican en el diseño de las computadoras.

Proposición. Los conjuntos unitarios {↓} y {|} son conjuntos adecuados de conectivas: toda función de verdad puede expresarse mediante una forma enunciativa en la que solo aparece la conectiva ↓, o solo aparece la conectiva **|**.

Argumentaciones

Retomamos lo que vimos anteriormente, en cuanto a la importancia de la "forma" del razonamiento por encima del significado de los enunciados que intervienen, partiendo de un par de definiciones:

Definición. Forma argumentativa (o argumentación). Una *forma argumentativa* es una sucesión finita de formas enunciativas, de las cuales la última se considera como la *conclusión* de las anteriores, conocidas como *premisas*. La notación es:

$$A_1, A_2, \ldots, A_n \therefore A$$

Para que una forma argumentativa sea válida debe representar un razonamiento correcto. Es decir, bajo cualquier asignación de valores de verdad a las variables de enunciado, si las premisas A_1, A_2, …, A_n, toman el valor V, la conclusión A también debe tomar el valor V. Precisando:

Definición. Forma argumentativa válida. Una forma argumentativa $A_1, A_2, \ldots, A_n \therefore A$ es *inválida* si es posible asignar valores de verdad a las variables de enunciado que aparecen en ella, de tal manera que A_1, A_2,…, A_n, tomen el valor V y A tome el valor F. De lo contrario la forma argumentativa es *válida*.

Por ejemplo, analicemos la validez de la siguiente argumentación:

> Si Alexia toma el autobús, entonces Alexia pierde su entrevista si el autobús llega tarde.
>
> Alexia no vuelve a su casa, si Alexia pierde su entrevista y Alexia se siente deprimida.
>
> Si Alexia no consigue el trabajo, entonces Alexia se siente deprimida y Alexia no vuelve a su casa.
>
> Por lo tanto, si Alexia toma el autobús entonces Alexia no consigue el trabajo si el autobús llega tarde.

En primer lugar debemos construir la forma argumentativa correspondiente, traduciendo del lenguaje natural al lenguaje simbólico. Consideraremos las siguientes variables de enunciado:

p: Alexia toma el autobús
q: Alexia pierde su entrevista
r: el autobús llega tarde
s: Alexia vuelve a su casa
t: Alexia se siente deprimida
u: Alexia consigue el trabajo

Construyamos las premisas del razonamiento anterior:

A_1: $p \to (r \to q)$
A_2: $(q \land t) \to (\neg s)$
A_3: $(\neg u) \to (t \land (\neg s))$

La conclusión es:

A: $p \to (r \to (\neg u))$

Tenemos entonces la forma argumentativa A_1, A_2, A_3 $\therefore$ A. Observemos que es posible asignar valores de verdad a las variables de enunciado de modo tal que las premisas tomen el valor V y la conclusión el valor F:

A_1	**A_2**	**A_3**	**A**
$p \to (r \to q)$	$(q \land t) \to (\neg s)$	$(\neg u) \to (t \land (\neg s))$	$p \to (r \to (\neg u))$
V **V** V V V	V F F **V** FV	FV **V** FF FV	V **F** V F FV

Así, la forma argumentativa es inválida. Si en cambio modificamos la conclusión de la siguiente manera obtenemos una forma argumentativa válida:

Por lo tanto, Alexia consigue el trabajo si Alexia pierde su entrevista y Alexia vuelve a su casa.

Nos queda:

A: $(q \land s) \to u$

En este caso es imposible asignar valores de verdad a las variables de enunciado tal que las premisas sean verdaderas y la conclusión falsa. Independientemente de otras variables de enunciado, sería el caso de asignar el valor F a la variable u (y en consecuencia el valor V a la variable s), para que A sea falsa, pero como vemos en la tabla de abajo, para que A_3 sea verdadera con la variable u falsa la variable s debe ser falsa (absurdo):

A_1	A_2	A_3	A
$p \rightarrow (r \rightarrow q)$	$(q \wedge t) \rightarrow (\neg s)$	$(\neg u) \rightarrow (t \wedge (\neg s))$	$(q \wedge s) \rightarrow u$
		V F **V** V V V F	V V V **F** F

La variable s debe tomar el valor F y al mismo tiempo el valor V (absurdo).

Es interesante remarcar cómo la traducción del lenguaje natural al lenguaje simbólico nos ha permitido desligarnos del significado de las palabras. Intuitivamente hubiésemos pensado que la primera conclusión (si Alexia toma el autobús, entonces Alexia no consigue el trabajo si el autobús llega tarde) era válida, mientras que la segunda (Alexia consigue el trabajo si Alexia pierde su entrevista y Alexia vuelve a su casa) no lo era, debido a nuestra idea acerca de las implicancias de llegar tarde o perder una entrevista de trabajo. La traducción al lenguaje simbólico nos permitió analizar matemáticamente la validez del razonamiento, basándonos en su estructura y no en su significado. Enseguida veremos aproximaciones para mecanizar los razonamientos.

Proposición. La forma argumentativa $A_1, A_2, \ldots, A_n \therefore A$ es válida si y solo si la forma enunciativa $(A_1 \wedge A_2 \wedge \ldots \wedge A_n) \rightarrow A$ es una tautología (es decir, si y solo si la conjunción de las premisas implican lógicamente a la conclusión).

Para referirnos a formas argumentativas válidas utilizamos la siguiente notación:

$$\varphi \models A$$

que se lee: "φ implica lógicamente a A" o "A se deduce de φ", siendo φ un conjunto de premisas.

Mecanismos formales de razonamiento

Un mecanismo formal de razonamiento (o de inferencia, deducción, demostración) consiste en una colección de reglas que pueden ser aplicadas sobre cierta información inicial para derivar información adicional, en una forma puramente sintáctica. A continuación se presentan

dos mecanismos formales estándar de razonamiento para la lógica proposicional. En primer lugar describimos un *sistema axiomático* (o *deductivo*) llamado L, y luego un sistema sin axiomas, conocido como *deducción natural*.

Sistema axiomático L

Nuestra primera aproximación de mecanismo formal de razonamiento es un sistema axiomático denominado L (ya antes definimos el alfabeto y la gramática). Un sistema axiomático está compuesto por un conjunto de *axiomas* (en realidad esquemas de axiomas) y un conjunto *de reglas de inferencia*. Los axiomas son fórmulas bien formadas. Las reglas determinan qué fórmulas pueden inferirse a partir de qué fórmulas. Por ejemplo, una regla de inferencia clásica es el *modus ponens*, según la cual a partir de las fórmulas A y $A \rightarrow B$ se puede inferir B.

Axiomas de L

Los axiomas de un sistema axiomático son un conjunto de fórmulas que se toman como punto de partida para las demostraciones. Un conjunto de axiomas muy conocido para la lógica proposicional es el que definió J. Lukasiewicz:

- $L_1 : A \rightarrow (B \rightarrow A)$
- $L_2 : (A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$
- $L_3 : ((\neg A) \rightarrow (\neg B)) \rightarrow (B \rightarrow A)$

Reglas de inferencia de L

Una regla de inferencia es una función que asigna una fórmula (conclusión) a un conjunto de fórmulas (premisas). Naturalmente la idea es que las reglas de inferencia transmitan la verdad de las premisas a la conclusión (también conocida como *teorema*), es decir que sea imposible alcanzar una conclusión falsa a partir de premisas verdaderas.

El sistema L tiene una única regla de inferencia, el modus ponens, ya referido previamente:

- MP: a partir de A y de $A \rightarrow B$ se infiere B

Tener en cuenta que A, B y C, mencionadas en los axiomas y la regla de inferencia de L, pueden ser sustituidas por cualquier fórmula bien formada. El modus ponens es una regla muy razonable desde el punto de vista intuitivo, corresponde a una de las maneras estándar de proceder en una argumentación en el lenguaje cotidiano. Por su parte los axiomas de L no resultan tan intuitivos, y no son los únicos posibles. Vistos como formas enunciativas son tautologías: obviamente es indispensable comenzar a razonar desde elementos verdaderos.

Deducción natural

Un sistema de lógica proposicional también puede definirse a partir de un conjunto vacío de axiomas. Se trata de la deducción natural. Sus reglas de inferencia intentan capturar el modo en que naturalmente razonamos acerca de las conectivas lógicas:

Introducción de la negación	De $A \rightarrow B$ y $A \rightarrow (\neg B)$ se infiere $\neg A$
Eliminación de la negación	De $\neg A$ se infiere $A \rightarrow B$
Eliminación de la doble negación	De $\neg (\neg A)$ se infiere A
Introducción de la conjunción	De A y B se infiere $A \wedge B$
Eliminación de la conjunción	De $A \wedge B$ se infiere A
	De $A \wedge B$ se infiere B
Introducción de la disyunción	De A se infiere $A \vee B$
Eliminación de la disyunción	De $A \vee B$, $A \rightarrow R$ y $B \rightarrow R$ se infiere R
Introducción del bicondicional	De $A \rightarrow B$ y $B \rightarrow A$ se infiere $A \leftrightarrow B$
Eliminación del bicondicional	De $A \leftrightarrow B$ se infiere $A \rightarrow B$
	De $A \leftrightarrow B$ se infiere $B \rightarrow A$
Eliminación del condicional	De A y $A \rightarrow B$ se infiere B
Introducción del condicional	Si A permite una prueba de B, se infiere $A \rightarrow B$

Demostración

Una *demostración* (o *prueba*) es una sucesión de aplicaciones de reglas de inferencia que permite llegar a una conclusión a partir de determinadas premisas o axiomas. Como la implicación de una fórmula a otra es una relación transitiva, la idea es que todas las fórmulas que se vayan obteniendo sucesivamente estén implicadas por las premisas o axiomas.

De esta manera las pruebas se pueden representar como las formas argumentativas estudiadas anteriormente: una sucesión finita de fórmulas bien formadas o formas enunciativas $A_1, A_2, \ldots, A_n$, tal que para todo i, con $1 \leq i \leq n$, A_i es una premisa o axioma, o bien se infiere de miembros anteriores de la sucesión como consecuencia directa de la aplicación de una regla de inferencia. Usaremos la notación:

$$\Gamma \vdash_{SD} A_n$$

que se lee: “A partir de los elementos del conjunto Γ se infiere A_n”. El subíndice SD especifica qué sistema deductivo se utiliza para llevar a cabo la prueba (por ejemplo SD puede ser L). Notar la diferencia entre los símbolos |– y |=. El primero, que estamos considerando ahora, se asocia a lo sintáctico, mientras que el segundo lo hemos empleado previamente para las definiciones semánticas.

Para ejemplificar una prueba utilizando L demostramos a continuación la implicación $A \to A$, es decir que vamos a llevar a cabo $\vdash_L A \to A$:

1. $(A \to ((A \to A) \to A)) \to ((A \to (A \to A)) \to (A \to A))$ instanciando el axioma L_2
2. $A \to ((A \to A) \to A)$ instanciando el axioma L_1
3. $(A \to (A \to A)) \to (A \to A)$ MP entre 1 y 2
4. $A \to (A \to A)$ instanciando el axioma L_1
5. $A \to A$ MP entre 3 y 4

Para un segundo ejemplo también en L tendremos en cuenta la siguiente argumentación:

Si tenemos una buena especificación entonces obtenemos un diseño correcto.

Si obtenemos un diseño correcto obtenemos un buen programa, a menos que nuestro programador sea mediocre.

Nuestro programador no es mediocre.

Se quiere demostrar:

Si tenemos una buena especificación obtenemos un buen programa.

Primeramente formalizamos el razonamiento mediante las variables de enunciado p, q, r, s:

p : Tenemos una buena especificación

q : Obtenemos un diseño correcto

r : Obtenemos un buen programa

s : Nuestro programador es mediocre

Las fórmulas que representan la argumentación son:

$A_1 : p \to q$

$A_2 : q \to ((\neg s) \to r)$

$A_3 : \neg s$

$A_4 : p \to r$

La premisa A_2 también se podría formalizar con la fórmula $(q \wedge (\neg s)) \to r$, que es lógicamente equivalente a la que hemos planteado.

Como primer paso, utilizando las premisas A_1 y A_2, los axiomas L_1 y L_2 y la regla de modus ponens, no es difícil alcanzar la siguiente conclusión intermedia (el detalle de los pasos queda como ejercicio para el lector):

$A_5 : p \to ((\neg s) \to r)$

Por instanciación del axioma L_2 obtenemos:

$A_6 : (p \to ((\neg s) \to r)) \to ((p \to (\neg s)) \to (p \to r))$

Aplicando modus ponens entre A_5 y A_6 :

$A_7 : (p \to (\neg s)) \to (p \to r)$

Por instanciación del axioma L_1 obtenemos:

$A_8 : (\neg s) \to (p \to (\neg s))$

Aplicando modus ponens entre A_3 y A_8 :

$A_9 : p \to (\neg s)$

Finalmente, aplicando nuevamente modus ponens, ahora entre A_7 y A_9, llegamos a la conclusión buscada:

$A_{10} : p \to r$

Sensatez y completitud de un sistema deductivo

De un sistema deductivo se espera naturalmente que sus demostraciones produzcan conclusiones correctas, y también en lo posible la propiedad inversa, es decir que sea capaz de demostrar todas y cada una de ellas. Formalmente ambas propiedades se pueden formular de la siguiente manera:

- Un sistema deductivo SD es *sensato* (también se lo conoce como *correcto*) si $\Gamma \vdash_{SD} A$ implica $\Gamma \models A$.
- Recíprocamente, un sistema deductivo SD es *completo* si $\Gamma \models A$ implica $\Gamma \vdash_{SD} A$.

Por un simple razonamiento inductivo es claro que la sensatez de un sistema deductivo está garantizada si se cuenta con axiomas verdaderos y reglas de inferencia sensatas, es decir que preservan la verdad. Por su parte, la completitud depende de los conjuntos de axiomas y reglas de inferencia que se escojan (por ejemplo, solo con la regla de modus ponens un sistema deductivo no podría inferir una conclusión que proceda de un razonamiento del tipo *modus tollens*, que establece que de $A \to B$ y $\neg B$ se infiere $\neg A$).

Proposición. El sistema axiomático L es sensato y completo.

Decidibilidad

Un sistema deductivo completo puede demostrar cualquier conclusión lógicamente implicada por las premisas, lo que hemos anotado con $\Gamma \models A$. ¿Pero qué ocurre si $\Gamma \not\models A$?

En la teoría de la computación, se define un problema de decisión como aquél que tiene dos respuestas posibles: "sí" o "no". Se dice que un problema de decisión es *decidible* si existe un algoritmo (es decir, un procedimiento que siempre termina) que lo resuelve.

En el marco de los sistemas deductivos es muy relevante también la cuestión de la decidibilidad, que se formula de la siguiente manera. Dados Γ y A, ¿existe algún algoritmo que responda "sí" en el caso de que $\Gamma \models A$ y que responda "no" en el caso de que $\Gamma \not\models A$?

Claramente, en la lógica proposicional esta cuestión tiene una respuesta positiva, utilizando algoritmos basados en las tablas de verdad.

Limitaciones de la lógica proposicional

Podemos observar que en el universo del discurso de esta lógica no hay objetos, sino afirmaciones que se formulan sobre objetos. Esto hace que el poder expresivo, y por tanto la utilidad de esta lógica resulten pobres. En los razonamientos deductivos hay normalmente premisas que expresan conocimiento sobre objetos, lo que es imposible formalizar en la lógica proposicional a menos, claro está, que forcemos la conceptualización particularizando lo general, tal como hemos hecho en los ejemplos de este capítulo. En el siguiente capítulo estudiaremos una lógica más expresiva, la lógica de predicados, que nos permitirá representar relaciones entre objetos.

Ejercicios

1. Traduzca al lenguaje simbólico los siguientes enunciados:
   i. Juan necesita un matemático o un informático.
   ii. Si Juan necesita un informático entonces necesita un matemático.
   iii. Si Juan no necesita un matemático entonces necesita un informático.
   iv. Si Juan contrata un informático entonces el proyecto tendrá éxito.
   v. Si el proyecto no tiene éxito entonces Juan no ha contratado un informático.
   vi. El proyecto tendrá éxito si y solo si Juan contrata un informático.
   vii. Para aprobar Lógica, el alumno debe asistir a clase, desarrollar un cuaderno de prácticas aceptable y demostrar que dicho cuaderno ha sido desarrollado por él; o desarrollar un cuaderno de prácticas aceptable y aprobar el examen final.
   viii. El alumno puede asistir a clase u optar por un examen libre.

ix. Seleccione de la lista anterior un par de enunciados que tengan la misma forma, y un par de enunciados que tengan el mismo significado.

2. Dadas las letras de las siguientes canciones, simbolice los enunciados correspondientes. Además encuentre en los textos, de ser posible, contradicciones y tautologías:

i*. “Ya no sé qué hacer conmigo”. Cuarteto de Nos.*

Ya viajé, ya pegué, ya sufrí, ya eludí, ya huí, ya asumí, ya me fui, ya volví, ya fingí, ya mentí,
y entre tantas falsedades, muchas de mis mentiras ya son verdades.
Hice fácil las adversidades, y me compliqué en las nimiedades,
y oigo una voz que dice con razón,
vos siempre cambiando ya no cambiás más,
y yo estoy cada vez más igual,
ya no sé qué hacer conmigo.

ii. *“Un poco perdido”. Tan biónica.*

Todos los días tienen sol y tormenta,
si pudieras volver a confiar.
Anoche resucitaron los inmortales,
se escucha nuestra orquestita en los arrabales.

iii. *“Amores que matan”. Joaquín Sabina.*

No me esperes a las doce en el juzgado,
no me digas "volvamos a empezar",
yo no quiero ni libre ni ocupado,
ni carne ni pecado,
ni orgullo ni piedad.
Yo no quiero contigo ni sin ti,
lo que yo quiero, muchacha de ojos tristes,
es que mueras por mí.
Y morirme contigo si te matas,
y matarme contigo si te mueres,
porque el amor cuando no muere mata,
porque amores que matan nunca mueren.

3. Sean A , B, C y D formas enunciativas. Se sabe que $A \rightarrow B$ es una contradicción y que $C \rightarrow D$ es una tautología. Determinar, si es posible, cuáles de las siguientes formas enunciativas son tautologías y cuáles contradicciones. Justificar las respuestas.

i. $(C \rightarrow B) \vee (D \rightarrow B)$

ii. $(A \to C) \lor (B \to D)$

iii. $(A \to D) \land (B \to D)$

4. Evalúe la validez de la siguiente argumentación. Para ello escriba una forma argumentativa y determine si es válida o inválida:

    Si Superman fuese capaz de destruir el mal y si quisiese hacerlo entonces lo haría.

    Si Superman no fuese capaz de destruir el mal entonces no sería poderoso.

    Si Superman no quisiese destruir el mal entonces sería maligno.

    Superman no destruye el mal.

    Si Superman existe entonces es poderoso y no es maligno.

    Por lo tanto, Superman no existe.

5. Se sabe que:

    La página web tiene un error o el examen de álgebra no es el 2 de julio.

    Si el examen de álgebra es el 2 de julio entonces la página web tiene un error.

    El examen de álgebra es el 14 de julio si y solo si la página web tiene un error y el período de exámenes no termina el 10 de julio.

    Teniendo en cuenta que el período de exámenes termina el 10 de julio y que la página web tiene un error, deducir la verdad o falsedad de los siguientes enunciados:

    i. El examen de álgebra es el 2 de julio.

    ii. Si la página web no tiene un error entonces el examen de álgebra es el 14 de julio.

6. Se tienen las siguientes premisas:

    Si Juan tiene suerte y llueve entonces estudia.

    Juan aprobará si y solo si estudia o tiene suerte.

    Si Juan no tiene suerte entonces no llueve.

    Sabiendo que llueve, responder:

    i. ¿Aprobará Juan?

    ii. ¿Tendrá suerte Juan?

7. La isla de los caballeros y los pícaros está habitada solamente por estos dos tipos de personas. Los caballeros tienen la particularidad de que solo dicen la verdad, mientras que los pícaros siempre mienten. Hay dos personas, A y B, habitantes de la isla.

    i. A hace la siguiente afirmación: “Al menos uno de nosotros es pícaro”. ¿Qué son A y B?

    ii. A dice: “Soy un pícaro pero B no lo es”. ¿Qué son A y B?

    iii. Alguien pregunta a B: “¿Es usted un caballero?”. B responde: “Si soy un caballero entonces me comeré el sombrero”. Probar que B deberá comerse el sombrero.

8. En una demostración presentada en el capítulo en el marco del sistema axiomático L, se indicó que de las premisas:

$p \rightarrow q$

$q \rightarrow ((\neg s) \rightarrow r)$

podía llegarse sin dificultad a la conclusión:

$p \rightarrow ((\neg s) \rightarrow r)$

recurriendo a los axiomas L_1 y L_2 y la regla de modus ponens. Se pide desarrollar la prueba.

Bibliografía

Enderton, H. (2001). *A mathematical introduction to logic (2nd Edition).* Boston, MA, Academic Press, ISBN 978-0-12-238452-3.

Hamilton, A.G. (1988). *Logic for Mathematicians (2nd Edition).* Cambridge, Cambridge University Press, ISBN 978-0-521-36865-0.

Mendelson, E. (1997). *Introduction to Mathematical Logic (4th Edition).* Published by Chapman & Hall, 2-6 Boundary Row, London SEl 8HN, UK.

CAPÍTULO 2
Lógica de predicados

Claudia Pons

Introducción

La maquinaria de la lógica proposicional permite formalizar y teorizar sobre la validez de una gran cantidad de enunciados. Sin embargo existen enunciados intuitivamente válidos que no pueden ser probados por dicha lógica. Por ejemplo, considérese el siguiente razonamiento:

Todos los hombres son mortales.
Sócrates es un hombre.
Por lo tanto, Sócrates es mortal.

En este caso no existe ninguna de las conectivas estudiadas en la lógica proposicional. Su formalización es la siguiente:

p
q
Por lo tanto, r

Esta es claramente una forma de razonamiento incorrecto, lo que contradice nuestra intuición. Sucede que para estudiar la validez de este tipo de razonamientos necesitamos analizar la estructura interna de las variables de enunciado. Dicha problemática la resuelve la lógica de predicados.

Un *predicado* es “lo que se afirma de un sujeto en una proposición” (D.R.A.E.). Los predicados pueden definir propiedades sobre uno, dos, o más individuos (u objetos), establecen relaciones entre ellos. Así, hay predicados unarios (o monádicos), binarios, etc. Los predicados unarios definen relaciones de grado uno, como por ejemplo “el 7 es un número primo”. Un ejemplo de predicado binario, que define una relación de grado dos, es “el 9 es múltiplo de 3”.

Dicho de otra manera, la lógica de predicados nos permite “entrar en el contenido de las proposiciones”. Enunciados como “8 es menor que 10” y “10 es mayor que 8”, en la lógica

proposicional solo pueden representarse como elementos atómicos. De este modo no podríamos expresar ideas tan sencillas como: “si x es menor que y entonces y es mayor que x”. La lógica de predicados en cambio nos permite representar las relaciones “menor que” y “mayor que”.

Ahora bien, a veces también surge la necesidad de representar propiedades de relaciones, es decir relaciones entre relaciones. La lógica de predicados limitada a representar relaciones entre objetos se denomina *de primer orden*, la que permite expresar relaciones entre relaciones se conoce como *de segundo orden*, y así sucesivamente. En este capítulo nos centraremos en la lógica de predicados de primer orden.

Dominios

Tal como hicimos cuando presentamos la lógica proposicional, se van a describir enseguida los aspectos sintácticos y semánticos de la lógica de predicados. Antes, introducimos un componente central para su estudio, que es el *dominio*. Los dominios están constituidos por:

- Un *universo del discurso* U, que es un conjunto no vacío de objetos.

- Un conjunto finito F de *funciones*, cada una de las cuales asigna a una determinada cantidad de objetos o argumentos de U un objeto de U:

  $$F = \{f^1_1, f^1_2, \ldots, f^2_1, f^2_2, \ldots\}$$

  El símbolo f^i_j denota la j-ésima función de i argumentos.

- Un conjunto finito P de *relaciones* entre los objetos de U:

  $$P = \{P^1_1, P^1_2, \ldots, P^2_1, P^2_2, \ldots\}$$

  El símbolo P^i_j denota la j-ésima relación de grado i.

Las definiciones de las funciones y relaciones pueden ser e*xtensionales*, cuando se efectúan enumerando las tuplas que las componen, o *intensionales*, si se establecen a partir de otras funciones o relaciones. Precisando un poco más, una definición extensional de una función f de grado n, es decir $f : U^n \rightarrow U$, hace corresponder a n-tuplas de objetos de U con objetos de U. Y una definición extensional de una relación de grado n es un subconjunto del producto cartesiano U^n. Las definiciones extensionales expresan conocimiento sobre situaciones particulares de los objetos de U (conocimiento factual). En cambio, las definiciones intensionales expresan conocimiento normativo sobre el dominio.

Clarificamos los conceptos anteriores mediante el siguiente ejemplo:

- U = N, es decir que U es el conjunto de los números naturales.

- F = {suc, +}, es decir que F es el conjunto formado por la función suc (por sucesor) y la función +. Asumiendo definida la suma habitual de los números naturales, definimos la función suc del siguiente modo:

$$suc = \{(x, y) \mid y = x + 1\}$$

- P = {=, $\leq$}, es decir que P es el conjunto que tiene las relaciones habituales = y $\leq$ de los números naturales, que asumimos definidas.

Sea ahora este otro ejemplo, asumiendo definida la relación habitual > de los números naturales:

- U = {Alex, Tomás, Catty, Florencia}. U es un conjunto de niños.

- F = {edad, altura}, siendo:
  edad = {(Alex, 13), (Tomás, 15), (Catty, 14) , (Florencia, 15)}
  altura = {(Alex,174), (Tomás,176), (Catty, 168), (Florencia, 164)}

- P = {juega-básquet, toca-piano, más-alto, más-joven}, siendo:
  juega-básquet = {Tomás}
  toca-piano = {Alex, Catty}
  más-alto = {(x, y) | altura(x) > altura(y)}
  más-joven = {(x, y) | edad(y) > edad(x)}

Obsérvese que las nociones de verdad y falsedad, que en la lógica proposicional se declaran explícitamente o se calculan, en la lógica de predicados están directamente implícitas en el dominio. Más claro, tomando un caso del último ejemplo: en lugar de establecer la verdad de que Alex toca el piano, incluimos directamente a Alex en el conjunto de niños que tocan el piano, que es la manera extensional de definir dicha propiedad.

Sintaxis: el lenguaje simbólico de la lógica

Al igual que en el capítulo anterior, describimos primeramente el lenguaje simbólico que utilizaremos para estudiar los mecanismos de razonamiento, ahora de la lógica de predicados.

Alfabeto

El alfabeto del lenguaje está formado por:

- Un conjunto de símbolos de constantes $C = \{c_1, c_2, \ldots\}$.
- Un conjunto de símbolos de variables $X = \{x_1, x_2, \ldots\}$.
- Un conjunto de símbolos de funciones $F = \{f^1_1, f^1_2, \ldots, f^2_1, f^2_2, \ldots\}$.
- Un conjunto de símbolos de predicados $P = \{P^1_1, P^1_2, \ldots, P^2_1, P^2_2, \ldots\}$.
- Símbolos de conectivas (los mismos de la lógica proposicional): $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$.
- Paréntesis de apertura y cierre.
- El cuantificador universal $\forall$ ("para todo") y el cuantificador existencial $\exists$ ("existe").

Gramática

La gramática del lenguaje define dos clases de elementos, por un lado los *términos*, que son las expresiones que denotan los objetos del dominio, y por el otro las *fórmulas bien formadas*, con las que se expresan las relaciones entre los objetos.

Los términos se definen inductivamente de la siguiente manera:

i. Los símbolos de constantes y de variables son términos.
ii. Si $t_1, \ldots, t_n$ son términos y f^n_i es un símbolo de función, entonces $f^n_i(t_1, ..., t_n)$ es un término.
iii. Solo las expresiones que pueden ser generadas mediante las cláusulas i y ii en un número finito de pasos son términos.

Por su parte, las fórmulas bien formadas se definen así:

i. Si $t_1, \ldots, t_n$ son términos y P^n_i es un símbolo de predicado, entonces $P^n_i(t_1, ..., t_n)$ es una formula bien formada. En este caso se denomina *fórmula atómica* o directamente *átomo*.
ii. Si A y B son fórmulas bien formadas, entonces $(\neg A)$, $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$ y $(A \leftrightarrow B)$ también lo son.
iii. Si A es una fórmula bien formada y x es un símbolo de variable, entonces $(\forall x)\ A$ y $(\exists x)\ A$ son fórmulas bien formadas.
iv. Solo las expresiones que pueden ser generadas mediante las cláusulas i a iii en un número finito de pasos son fórmulas bien formadas.

Se mantiene la convención de poder prescindir de los paréntesis innecesarios. Los cuantificadores tienen un *alcance* o *radio de acción* determinado. Por ejemplo, en la fórmula bien formada:

$$(\forall x)\ (P^1_1(x) \rightarrow P^1_2(x))$$

las dos ocurrencias del símbolo de variable x suceden dentro del alcance del cuantificador $\forall$, mientras que en la fórmula bien formada:

$$(\forall x)\ P^1_1(x) \rightarrow P^1_2(x)$$

el $\forall$ alcanza solo a la primera ocurrencia de x. Diremos en el primer caso que x está *ligada*, y en el segundo que la primera x está *ligada* y la otra está *libre* (por lo que podría sustituirse por cualquier otro símbolo de variable). Una fórmula es *abierta* si contiene algún símbolo de variable libre, y *cerrada* si todos los símbolos de variables están ligados.

Veamos un par de ejemplos de uso del lenguaje de la lógica de predicados, considerando dominios concretos. Entraremos ya, por ahora informalmente, en el campo de la semántica. Para las expresiones de la aritmética utilizaremos los siguientes símbolos:

- c_1 será el símbolo de constante para representar el cero.
- f^1_1 será el símbolo de función para representar el sucesor.
- f^2_1 será el símbolo de función para representar la suma.
- f^2_2 será el símbolo de función para representar la multiplicación.
- P^2_1 será el símbolo de predicado para representar la igualdad.

Ejemplos de representaciones de propiedades aritméticas son:

- $\neg\ P^2_1\ (f^1_1\ (c_1), c_1)$. El sucesor del cero no es igual a cero.
- $(\forall x)\ P^2_1\ (f^2_1\ (x, c_1), x)$. El cero es el neutro de la suma.
- $(\forall x)(\forall y)\ P^2_1\ (f^2_1\ (x, y), f^2_1\ (y, x))$. La suma es conmutativa.
- $(\forall x)(\forall y)\ P^2_1\ (f^2_1\ (x, f^1_1\ (y)), f^1_1\ (f^2_1\ (x, y)))$. La suma de *x* y el sucesor de *y* es igual al sucesor de la suma de *x* e *y*.
- $(\forall x)\ P^2_1\ (f^2_2\ (x, c_1), c_1)$. Todo número multiplicado por cero da cero.
- $(\forall x)\ P^2_1\ (f^2_2\ (x, f^1_1\ (c_1)), x)$. El uno es el neutro de la multiplicación.

Como segundo ejemplo volvemos a un razonamiento sobre nacionalidades del capítulo anterior. El universo del discurso es el conjunto de todas las personas, y supongamos que c_1 representa al objeto Juan, y P^1_1 y P^1_2 a las propiedades “es mendocino” y “es argentino”, respectivamente. De esta manera podemos formular lo siguiente:

- $(\forall x)\ (P^1_1\ (x) \rightarrow P^1_2\ (x))$. Todos los mendocinos son argentinos.
- $P^1_1\ (c_1)$. Juan es mendocino.
- $P^1_2\ (c_1)$. Juan es argentino.

Semántica: interpretación y satisfacción

Como antes, para definir la semántica nos apoyamos fundamentalmente en los conceptos de interpretación y satisfacción. De todos modos ahora tendremos que añadir otros conceptos, por ejemplo el de valoración, que introduciremos enseguida.

Definición. Interpretación. Dados un lenguaje y un dominio, una *interpretación* I es una función que hace corresponder a los elementos del lenguaje con los elementos del dominio, satisfaciendo las siguientes condiciones:

- Si c_i es un símbolo de constante, entonces $I(c_i) \in U$ (los símbolos de constantes representan objetos del universo del discurso).
- Si f^n_i es un símbolo de función de grado n, entonces $I(f^n_i) = U^n \to U$ (los símbolos de función representan funciones del dominio).
- Si P^n_i es un símbolo de predicado de grado n, entonces $I(P^n_i) \subseteq U^n$ (los símbolos de predicado representan relaciones del dominio).

Por lo tanto, una interpretación formaliza la noción de que los símbolos representan las abstracciones de la realidad modelada en el dominio correspondiente. Retomemos el ejemplo de la aritmética. Hemos definido los símbolos c_1, f^1_1, f^2_1, f^2_2 y P^2_1. Lo que hicimos antes intuitivamente ahora podemos formalizarlo de la siguiente manera:

- $I(c_1)$ es el cero de los naturales.
- $I(f^1_1)$ es la función sucesor en los naturales.
- $I(f^2_1)$ es la función suma en los naturales.
- $I(f^2_2)$ es la función multiplicación en los naturales.
- $I(P^2_1)$ es la relación de igualdad en los naturales.

Bajo esta interpretación hemos formulado algunas propiedades, como:

- $(\forall x)\ P^2_1\ (f^2_1\ (x, c_1), x)$. El cero es el neutro de la suma, es decir: $(\forall x)(x + 0 = x)$.
- $(\forall x)(\forall y)\ P^2_1\ (f^2_1\ (x, y), f^2_1\ (y, x))$. La suma es conmutativa, es decir: $(\forall x)(\forall y)(x + y = y + x)$.

Si bien ésta es la interpretación "habitual" o "estándar", nada, salvo el sentido común, nos impide plantear otras interpretaciones de los símbolos. Por ejemplo, podríamos establecer:

$I(f^2_1)$ es la función potencia en los naturales, con potencia$(x, y) = x^y$

Bajo esta nueva interpretación, entonces, quedaría formulado lo siguiente, respectivamente:

- $(\forall x)\ P^2_1\ (f^2_1\ (x, c_1), x)$. El cero es el neutro de la potencia, es decir: $(\forall x)(x^0 = x)$, lo cual no se cumple en los números naturales.
- $(\forall x)(\forall y)\ P^2_1\ (f^2_1\ (x, y), f^2_1\ (y, x))$. La potencia es conmutativa, es decir: $(\forall x)(\forall y)(x^y = y^x)$, que tampoco se cumple en el dominio considerado.

Definición. Valoración. Para completar la definición anterior falta establecer una manera de asignar objetos a todos los términos del lenguaje. Esto se logra por medio de una función denominada *valoración* (en una interpretación). Dada una interpretación **I** y una valoración v, esta última queda completamente especificada indicando cómo se asignan objetos a los símbolos de variables, es decir $v(x_1)$, $v(x_2)$, …, dado que se define:

- Si c_i es un símbolo de constante, $v(c_i) = \mathbf{I}(c_i)$.
- Si f^n_i es un símbolo de función, $v(f^n_i\ (t_1, ..., t_n)) = \mathbf{I}(f^n_i)(v(t_1), ..., v(t_n))$.

Por ejemplo, en la interpretación estándar de la aritmética, fijando v(x) = 7 la valoración del término $f^1_1\ (f^2_1\ (x, c_1))$ se obtiene de la siguiente manera:

$$v(f^1_1\ (f^2_1\ (x, c_1))) = \mathbf{I}(f^1_1)(v(f^2_1\ (x, c_1))) = suc(v(f^2_1\ (x, c_1))) = suc(\mathbf{I}(f^2_1)(v(x), v(c_1))) =$$
$$suc(+(v(x), v(c_1))) = suc(+(7, 0)) = suc(7) = 8$$

Definición. Satisfacción. En la lógica proposicional la satisfacción de un enunciado depende de la interpretación de las variables de enunciado (p, q, r, etc). Por lo visto recién, en la lógica de predicados tenemos que considerar también la valoración de los términos. Para denotar que una fórmula A se satisface con una interpretación **I** y una valoración v, escribiremos $|=_{I,v}$ A. Haciendo un parangón entre las variables de enunciado de la lógica proposicional y las fórmulas atómicas de la lógica de predicados (extensible respectivamente a los enunciados y las fórmulas de primer orden en general): en el primer caso, a las variables de enunciado se les asigna el valor verdadero o falso; en el segundo caso, una fórmula atómica se satisface o es verdadera con una interpretación y una valoración, si luego de evaluarse sus términos e interpretarse su símbolo de predicado, se obtiene una tupla de objetos de la relación representada. La definición inductiva de la satisfacción de una fórmula es la siguiente:

- $|=_{I,v} P(t_1,..,t_n)$ si y solo si $(v(t_1), ...,v(t_n)) \in \mathbf{I}(P)$
- $|=_{I,v} \neg A$ si y solo si no es el caso que $|=_{I,v} A$
- $|=_{I,v} A \vee B$ si y solo si o bien $|=_{I,v} A$ o bien $|=_{I,v} B$ o ámbos
- $|=_{I,v} A \wedge B$ si y solo si $|=_{I,v} A$ y $|=_{I,v} B$
- $|=_{I,v} A \rightarrow B$ si y solo si no es el caso que $|=_{I,v} A$ y no $|=_{I,v} B$

- $|=_{I,v} A \leftrightarrow B$ si y solo si $|=_{I,v} A \rightarrow B$ y $|=_{I,v} B \rightarrow A$
- $|=_{I,v} (\forall x)$ A si y solo si para todo objeto c de U se cumple $|=_{I,w}$ A, con w(x) = c, w(y) = v(y) para y ≠ x. Es decir que A es verdadera cualquiera sea la valoración de x.
- $|=_{I,v} (\exists x)$ A si y solo si para algún objeto c de U se cumple $|=_{I,w}$ A, con w(x) = c, w(y) = v(y) para y ≠ x. En este caso en cambio alcanza con que A sea verdadera considerando una valoración particular de x.

Definición. Verdad, modelo y validez. De acuerdo a lo que definimos recién, una fórmula A es *satisfactible* cuando existe una interpretación I y una valoración v que cumplen $|=_{I,v}$ A; de lo contrario es *insatisfactible*. Un conjunto de fórmulas $\{A_1, ..., A_n\}$ es satisfactible cuando existen I y v tales que $|=_{I,v} (A_1 \wedge ... \wedge A_n)$, e insatisfactible en caso contrario. Un ejemplo de conjunto insatisfactible es $\{P(x), \neg P(x)\}$, cualquiera sea P.

Si en particular una fórmula A se satisface con una interpretación I cualquiera sea la valoración utilizada, se dice que A es *verdadera* en I, y que I es un *modelo* de A. Se escribe así: $|=_I$ A. Una interpretación es un *modelo* de un conjunto de fórmulas si es un modelo de cada una de ellas. Por ejemplo, la interpretación estándar de la aritmética es un modelo de la fórmula ya vista $(\forall x)\ P^2_1\ (f^2_1\ (x, c_1), x)$: todo numero natural x cumple que x + 0 = x. Como contrapartida, decimos que una fórmula A es *falsa* en una interpretación I si no existe ninguna valoración en I que la satisfaga.

Finalmente, las fórmulas verdaderas en toda interpretación se identifican como *lógicamente válidas* o directamente *válidas*. La notación para una fórmula válida A es |= A. Por ejemplo, $(\forall x)\ P(x) \rightarrow (\exists x)\ P(x)$ es válida, cualquiera sea P. Particularmente, una fórmula válida cuya estructura se corresponde con una tautología de la lógica proposicional, como por ejemplo $P(x) \vee (\neg P(x))$, es una *tautología* de la lógica de predicados. Las fórmulas válidas no proporcionan información alguna de un dominio. Las fórmulas que buscamos para representar conocimiento son las verdaderas.

Mecanismos formales de razonamiento

Definimos previamente que un mecanismo de razonamiento para una lógica es un método sintáctico que se vale de reglas de inferencia para aplicarlas sobre cierta información inicial derivando nueva información. Nos permite ampliar el conocimiento que tenemos de un dominio, utilizando recursos meramente sintácticos, por supuesto mientras se cumpla la propiedad de sensatez que repasaremos enseguida. En un sentido representa el fin último de la lógica, para conducir nuestro pensamiento organizadamente a la construcción de juicios correctos, por medio del conocimiento aplicado adecuadamente. En lo que sigue describimos un conocido sistema axiomático de la lógica de predicados, que utilizaremos para estudiar los distintos aspectos de los mecanismos deductivos.

Sistema axiomático K

El sistema axiomático K incluye los (esquemas de) axiomas L_1, L_2 y L_3 y la regla MP (modus ponens) del sistema L que estudiamos en el marco de la lógica proposicional, y agrega tres (esquemas de) axiomas y una regla de inferencia, que se necesitan por la existencia de los cuantificadores. A los axiomas de L los renombramos con la letra K:

Axiomas de K

- $K_1 : A \rightarrow (B \rightarrow A)$
- $K_2 : (A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$
- $K_3 : ((\neg A) \rightarrow (\neg B)) \rightarrow (B \rightarrow A)$
- $K_4 : (\forall x)\ A \rightarrow A$, si x no está libre en A
- $K_5 : (\forall x)\ A(x) \rightarrow A(t)$, si t está libre para x en A(x)
- $K_6 : (\forall x)\ (A \rightarrow B) \rightarrow (A \rightarrow (\forall x)\ B)$, si x no está libre en A

El único axioma que amerita cierta aclaración es el K_5. Un término t está libre para x en A(x) si x no está libre en A(x) dentro del alcance de un ($\forall$y), siendo y integrante de t. La idea es que t puede sustituir cualquier ocurrencia libre de x en A(x) solo cuando no existe interacción con un cuantificador de A(x). Notar que este axioma es un caso general del axioma K_4, al que algunos autores no incluyen.

Reglas de inferencia de K

- MP: a partir de A y de $A \rightarrow B$ se infiere B
- Generalización: de A se infiere $(\forall x)\ A$

La aplicación de la regla de generalización tiene un efecto contrario al del uso de los axiomas K_4 y K_5, los cuales particularizan a partir de fórmulas cuantificadas universalmente.

Demostración

Mantenemos la definición del capítulo anterior. Una *demostración* (o *prueba*) es una sucesión de aplicaciones de reglas de inferencia que a partir de premisas o axiomas obtiene una conclusión (o *teorema*).

Desarrollamos a continuación un ejemplo muy sencillo de demostración en el marco del sistema axiomático K. Vamos a probar, utilizando nomenclatura previa:

P_1 : Todos los mendocinos son argentinos. Es decir: $(\forall x)(P^1_1(x) \rightarrow P^1_2(x))$.

P_2 : Juan es mendocino. Es decir: $P^1_1(c_1)$.

Conclusión: Juan es argentino. Es decir: $P^1_2(c_1)$.

La prueba es la siguiente:

1.	$(\forall x)(P^1_1(x) \rightarrow P^1_2(x))$	premisa P_1
2.	$(\forall x)(P^1_1(x) \rightarrow P^1_2(x)) \rightarrow (P^1_1(c_1) \rightarrow P^1_2(c_1))$	instanciando el axioma K_5
3.	$P^1_1(c_1) \rightarrow P^1_2(c_1))$	MP entre 1 y 2
4.	$P^1_1(c_1)$	premisa P_2
5.	$P^1_2(c_1)$	MP entre 3 y 4

Sensatez, completitud y decidibilidad de un sistema deductivo

Al igual que lo visto para la lógica proposicional, definimos:

- Un sistema deductivo SD es *sensato* (o *correcto*) si $\Gamma \vdash_{SD} A$ implica $\Gamma \models A$.
- Recíprocamente, un sistema deductivo SD es *completo* si $\Gamma \models A$ implica $\Gamma \vdash_{SD} A$.

Por lo tanto, considerando solo los axiomas y reglas de inferencia de SD, si SD es sensato se puede asegurar que solo permite demostrar fórmulas válidas, y si es completo, que toda fórmula válida es demostrable en SD.

Proposición. El sistema axiomático K es sensato y completo.

Fue K. Gödel quien probó en 1929 la completitud de la lógica de predicados de primer orden. Volveremos enseguida a este matemático y la problemática de la completitud.

Ahora bien, en el capítulo anterior sugeríamos como procedimiento alternativo a los sistemas deductivos para las demostraciones lógicas, algoritmos basados en las tablas de verdad. En la lógica de predicados, intuitivamente, una idea de este tipo, probando combinaciones de valores para los distintos componentes de las fórmulas, asoma como inaplicable en general: se trataría de determinar en tiempo finito que una fórmula sea válida, es decir verdadera para todas las (posiblemente infinitas) interpretaciones y todas las (posiblemente infinitas) valoraciones.

En efecto, en 1936 A. Church e independientemente A. Turing demostraron la indecidibilidad de esta lógica: no existe un algoritmo que pueda determinar, para todo Γ y A, si $\Gamma \models A$ o $\Gamma \not\models A$. Lo mejor que se puede tener es un algoritmo que responda "sí" cuando se cumple $\Gamma \models A$, por la completitud (por eso se dice que la lógica de predicados de primer orden es *semi-decidible*). Existen lógicas particulares, de alguna manera restringidas, que son

decidibles. Por ejemplo cuando el lenguaje no tiene símbolos de constantes ni de funciones y los símbolos de predicados son solo de un argumento.

Sistemas de primer orden

Ya hemos observado que sistemas deductivos como K son adecuados para fundamentar el mecanismo de razonamiento, pero insuficientes cuando se quiere profundizar en dominios particulares. Cada dominio debe tener naturalmente sus propios axiomas adicionales, fórmulas verdaderas solo en una cierta interpretación.

En consecuencia, se definen *sistemas de primer orden*, *extensiones* de axiomáticas como K que se obtienen ampliando o modificando el conjunto original de axiomas, de manera que toda fórmula que se puede probar a partir del esquema original se puede seguir probando, pudiendo demostrarse además nuevas fórmulas.

Describimos a continuación dos extensiones del sistema axiomático K muy conocidas y que resultan de gran utilidad: el *sistema de primer orden con igualdad*, y la *aritmética de primer orden* (a la que nos hemos referido en varias ocasiones).

Sistemas de primer orden con igualdad

La relación de igualdad está presente en la mayoría de los dominios. En nuestros ejemplos la hemos representado con el símbolo de predicado P^2_1, teniendo en cuenta la interpretación pretendida. Es claro que la fórmula $(\forall x)\ P^2_1(x, x)$ no es válida, porque P^2_1 admite otras interpretaciones que no la satisfacen.

La manera de asegurar que el símbolo de predicado que representa la relación de igualdad sea interpretado adecuadamente es agregando *axiomas de igualdad*. Los describimos a continuación, denominándolos E_1, E_2 y E_3:

- $E_1 : P^2_1(x, x)$
- $E_2 : P^2_1(t_k, u) \rightarrow P^2_1(f^n_i(t_1, \ldots, t_k, \ldots, t_n), f^n_i(t_1, \ldots, u, \ldots, t_n))$, siendo $t_1, \ldots, t_k, \ldots, t_n$ y u términos cualesquiera, y f^n_i cualquier símbolo de función.
- $E_3 : P^2_1(t_k, u) \rightarrow (P^n_i(t_1, \ldots, t_k, \ldots, t_n) \rightarrow P^n_i(t_1, \ldots, u, \ldots, t_n))$, siendo $t_1, \ldots, t_k, \ldots, t_n$ y u términos cualesquiera, y P^n_i cualquier símbolo de predicado.

Toda extensión del sistema axiomático K que incluye los axiomas de igualdad se conoce como *sistema de primer orden con igualdad*. A su vez, los modelos en los que el símbolo de predicado P^2_1 se interpreta como la relación de igualdad se denominan modelos *normales*.

Aritmética de primer orden

Se conoce como *aritmética de primer orden* al sistema de primer orden que se obtiene como extensión de la axiomática K, añadiendo los tres axiomas de igualdad y axiomas propios de la

aritmética, estos últimos basados en los postulados que formuló el matemático G. Peano en el siglo XIX para definir a los números naturales. El lenguaje se denota con L_N. Para facilitar la enunciación de los axiomas vamos a usar directamente los símbolos 0, +, . , para el cero, la suma y la multiplicación, respectivamente, el símbolo f para la función sucesor, y en todos los casos la notación infija en lugar de la prefija. Los axiomas propios de la aritmética son:

- $N_1 : (\forall x) \neg (f(x) = 0)$ — 1er axioma del sucesor
- $N_2 : (\forall x)(\forall y)(f(x) = f(y) \rightarrow x = y)$ — 2do axioma del sucesor
- $N_3 : (\forall x)(x + 0 = x)$ — 1er axioma de la suma
- $N_4 : (\forall x)(\forall y)(x + f(y) = f(x + y))$ — 2do axioma de la suma
- $N_5 : (\forall x) (x . 0 = 0)$ — 1er axioma de la multiplicación
- $N_6 : (\forall x)(\forall y)(x . f(y) = x . y + x)$ — 2do axioma de la multiplicación
- $N_7 : P(0) \rightarrow ((\forall x)(P(x) \rightarrow P(f(x))) \rightarrow (\forall x) P(x))$, para toda fórmula P(x) en que ocurre libre x — axioma de inducción matemática

El axioma N_7 se basa en el postulado de Peano que establece que para todo conjunto A de números naturales, si $0 \in A$ y si $f(n) \in A$ siempre que $n \in A$, entonces A contiene a todos los números naturales. Como el lenguaje L_N es de primer orden, el axioma N_7 no puede representar adecuadamente la expresión "para todo conjunto A de números naturales", porque necesitaría un cuantificador de segundo orden. De esta manera cada caso particular del axioma N_7 corresponde al postulado de Peano para un cierto conjunto particular. Más aún, el postulado se formula sobre un conjunto no numerable (conjuntos de números naturales), mientras que el axioma se refiere solo a una cantidad numerable de conjuntos.

Los símbolos del lenguaje L_N se pueden interpretar de la manera habitual (*modelo estándar*), o de otras maneras. Se demuestra que existen otros infinitos modelos (*no estándar* o *inintencionales*). Justamente, los postulados de Peano definen una noción más amplia de sucesión matemática que la concebida originalmente.

A diferencia de la lógica de predicados de primer orden, la aritmética de primer orden es incompleta, lo que también demostró Gödel (1931). En el último capítulo, fundamentalmente en sus últimas notas, se profundiza sobre este tópico, y en general sobre la sensatez, completitud y decidibilidad de los sistemas axiomáticos.

Completamos la sección con un ejemplo de prueba en la aritmética de primer orden. Vamos a probar: 1 + 1 = 2. Por lo tanto tendremos en cuenta los axiomas K_1 a K_6, E_1 a E_3 y N_1 a N_7, y las reglas MP y Generalización. En algunos casos abreviamos el término f(0) con 1:

1. $(\forall x)(x + 0 = x)$ — axioma N_3
2. $(\forall x)(x + 0 = x) \rightarrow 1 + 0 = 1$ — axioma K_5
3. $1 + 0 = 1$ — MP entre 1 y 2
4. $(\forall x)(\forall y)(x + f(y) = f(x + y))$ — axioma N_4
5. $(\forall x)(\forall y)(x + f(y) = f(x + y)) \rightarrow (\forall y)(1 + f(y) = f(1 + y))$ — axioma K_5

6.	$(\forall y)(1 + f(y) = f(1 + y))$	MP entre 4 y 5
7.	$(\forall y)(1 + f(y) = f(1 + y)) \rightarrow 1 + f(0) = f(1 + 0)$	axioma K_5
8.	$1 + f(0) = f(1 + 0)$	MP entre 6 y 7
9.	$x = y \rightarrow f(x) = f(y)$	axioma E_2
10.	$1 + 0 = 1 \rightarrow f(1 + 0) = f(1)$	se deriva de 9
11.	$f(1 + 0) = f(1)$	MP entre 3 y 10
12.	$(\forall x)(\forall y)(\forall z)(x = y \rightarrow (y = z \rightarrow x = z))$	teorema
13.	$1 + f(0) = f(1 + 0) \rightarrow (f(1 + 0) = f(1) \rightarrow 1 + f(0) = f(1))$	se deriva de 12
14.	$f(1 + 0) = f(1) \rightarrow 1 + f(0) = f(1)$	MP entre 8 y 13
15.	$1 + f(0) = f(1)$	MP entre 11 y 14

Como f(0) se abrevia con 1 y f(1) con 2, llegamos a 1 + 1 = 2. El teorema referido en el paso 12 se prueba empleando los axiomas de igualdad. Su demostración, como así también las derivaciones mencionadas en los pasos 10 y 13, quedan como ejercicio para el lector.

Ejercicios

1. Expresar en un lenguaje de primer orden el conocimiento asociado a las siguientes situaciones:
   i. Todo peluquero afeita a todo aquél que no se afeita a sí mismo. Ningún peluquero afeita a alguien que se afeite a sí mismo.
      Con el conocimiento disponible, ¿se puede deducir que los peluqueros no existen?
   ii. Ningún dragón que viva en un zoológico es feliz. Cualquier animal que encuentre gente amable es feliz. Las personas que visitan los zoológicos son amables. Los animales que viven en zoológicos encuentran personas que visitan zoológicos.
      Encontrar suposiciones adicionales que permitan concluir que ningún dragón vive en un zoológico.
   iii. Si alguien hace algo bueno, ese alguien es bueno. Del mismo modo, si alguien hace algo malo, es malo. Sebastián ayuda a su madre y también miente algunas veces. Mentir es malo y ayudar es bueno.
      Determinar si con el conocimiento disponible es posible deducir que Sebastián es bueno. ¿Y es posible deducir que es malo?
   iv. El Capitán Wine era responsable de la seguridad de sus pasajeros y su carga. Pero en su último viaje, se emborrachaba todas las noches y fue responsable de la pérdida del barco, con todo lo que llevaba. Se rumoreaba que estaba loco, pero los médicos lo encontraron responsable de sus actos. Usualmente, el capitán Wine no actuaba borracho. Durante aquel viaje, el capitán Wine se comportó muy irresponsablemente. El capitán Wine sostuvo que las tormentas fueron las responsables de la pérdida del barco, pero en el proceso que

se le siguió fue encontrado responsable de la pérdida de vidas y bienes. Todavía vive, y es responsable de la muerte de muchas mujeres y niños.

Con la información disponible, ¿es posible deducir que no siempre el capitán Wine se comportaba en forma responsable en las tormentas? Identificar los distintos significados de responsabilidad con diferentes predicados.

2. Dar interpretaciones para los siguientes lenguajes de primer orden, y traducir en cada caso las fórmulas presentadas a oraciones apropiadas en lenguaje natural.
   i. $\forall(x)\forall(y)(A^2_1 (x, y) \rightarrow A^2_1 (y, x))$
   $\forall(x)\ A^2_1 (y, x))$
   $\forall(x)\forall(y)\forall(z)(A^2_1 (x, y) \wedge A^2_1 (y, z) \rightarrow A^2_1 (x, z))$
   ii. $\forall(x)(A^2_1 (x, c) \rightarrow A^2_1 (x, f(y)))$
   $\forall(x) \neg A^2_1 (x, x)$
   $\neg \forall(x)\forall(y)\ A^2_1 (x, y)$

3. Determinar si las siguientes fórmulas escritas en algún lenguaje de primer orden son contradictorias, satisfactibles en alguna interpretación, verdaderas en alguna interpretación o lógicamente válidas. Fundamentar.
   i. $(\exists x)(\neg A^1_1 (x) \vee (\forall x)(A^1_1 (x) \vee B^1_1 (x)))$
   ii. $(\exists y)(\exists x)(A^2_1 (x, y) \rightarrow (\exists x)(\exists y)\ A^2_1 (x, y))$

4. Probar a partir de los axiomas de igualdad:
   i. $\forall(x)\ P^2_1 (x, x)$
   ii. $(\forall x)(\forall y)(P^2_1 (x, y) \rightarrow P^2_1 (y, x))$
   iii. $(\forall x)(\forall y)(\forall z)(P^2_1 (x, y) \rightarrow (P^2_1 (y, z) \rightarrow (P^2_1 (x, z)))$

5. Completar la prueba de la fórmula 1 + 1 = 2 desarrollada parcialmente en la sección dedicada a la aritmética de primer orden.

Bibliografía

Enderton, H. (2001). *A mathematical introduction to logic (2nd edition).* Boston, MA: Academic Press. ISBN 978-0-12-238452-3.

Fernández Fernández, G. (2004). *Representación del conocimiento en sistemas inteligentes. Parte II: Lógicas de base.* Universidad Politécnica de Madrid. En línea: http://dit.upm.es/~gfer/ssii/rcsi/

Hamilton, A.G. (1988). *Logic for Mathematicians (2nd edición).* Cambridge: Cambridge University Press, ISBN 978-0-521-36865-0.

Mendelson, E. (1997). *Introduction to Mathematical Logic (4ª edición).* Published by Chapman & Hall, 2-6 Boundary Row, London SEI 8HN, UK.

CAPÍTULO 3
Lógica modal

Clara Smith

Conceptos básicos

La lógica modal fue en sus orígenes (atribuidos a Aristóteles por la mayoría de los estudiosos) la lógica de *lo necesario* y *lo posible*. Más modernamente se la usó en el estudio de construcciones lingüísticas que califican las condiciones de validez de las proposiciones. Actualmente la lógica modal se aplica en el área de la informática para formalizar esquemas de razonamiento y sistemas donde intervienen múltiples agentes. La lógica modal en su versión proposicional es una extensión de la lógica proposicional, que también puede verse como un fragmento de la lógica de predicados con buenas propiedades computacionales, como la decidibilidad.

Una *modalidad* es una palabra o frase que puede aplicarse a una proposición $\mathcal{A}$ para crear una nueva proposición que hace una afirmación acerca del modo de verdad de $\mathcal{A}$ o de las circunstancias bajo las cuales $\mathcal{A}$ es verdadera: cuándo, dónde o cómo $\mathcal{A}$ es verdadera. Ejemplos son: "en el futuro sucederá $\mathcal{A}$" ($F\mathcal{A}$), "está permitido $\mathcal{A}$" ($\mathrm{P}\mathcal{A}$), "el agente sabe $\mathcal{A}$" ($\mathrm{K}\mathcal{A}$), "es necesario $\mathcal{A}$" ($\Box\mathcal{A}$), "alguna ejecución finita del programa π deja al sistema en un estado con información $\mathcal{A}$" ($<\pi>\mathcal{A}$), "es demostrable $\mathcal{A}$", entre muchas otras.

Lenguaje modal. Usamos un lenguaje proposicional clásico para trabajar. El lenguaje modal básico se funda sobre un conjunto numerable *P* de proposiciones usualmente denotadas con las letras p, q, r,… Expresiones complejas se forman sintácticamente del modo inductivo usual, usando (posiblemente) el operador $\bot$ (la constante *false*), el operador binario $\vee$ (disyunción), y el operador unario $\neg$ (negación). Como el comportamiento proposicional de esta lógica es clásico, asumimos que T (la constante *true*), $\wedge$ (conjunción), y $\rightarrow$ (condicional) se definen del modo esperado a partir de los símbolos ya provistos. A este lenguaje proposicional básico le agregamos un operador unario, que simbolizamos "◊" y llamamos coloquialmente "diamante" o "rombo". Con "◊" *modalizamos* las expresiones, decimos algo de ellas colocándoles un símbolo delante: ◊p.

Definición 3.1. Lenguaje. Las fórmulas bien formadas (fórmulas) del lenguaje modal básico $\mathcal{L}$ se definen a partir de un conjunto de variables proposicionales *P* = {p, q, r…}, con los operadores booleanos usuales, y con un operador unario ◊, del siguiente modo:

$$p \mid q \mid \ldots \mid \bot \mid \neg\mathcal{A} \mid \mathcal{A} \vee \mathcal{B} \mid \Diamond\mathcal{A}$$

con $\mathcal{A}$ y $\mathcal{B}$ fórmulas construidas del modo inductivo usual.

Agregamos el símbolo "□" para usarlo como una abreviatura. La relación entre ◊ y □ es *dual*: □p ≡ ¬◊¬p (el símbolo "≡" representa equivalencia lógica, lo mismo que "↔"). Tradicionalmente, □ se lee "es necesario" y ◊ se lee "es posible". Coloquialmente también llamamos "cuadrado" al "□".

Ejemplos de expresiones modales:

p → q si nos haces falta entonces te llamamos

p → ◊q si nos haces falta entonces es posible que te llamemos

p → q si queremos aprender entonces estudiamos

p → □q si queremos aprender entonces es necesario que estudiemos

Las lecturas de los símbolos □ y ◊, y también de otros símbolos modales, son muchas; diferentes lecturas de dichos símbolos ejercieron variada influencia a lo largo de los años en diferentes disciplinas, especialmente en la filosofía: se considera a la lógica modal como la herramienta por excelencia de la lógica filosófica, dando a los que la usan exquisitez y precisión para tratar con cuestiones metafísicas tales como la moral, para tratar con el tiempo, el espacio, el conocimiento, las obligaciones, etc. Algunos otros símbolos modales son, por citar algunos, O, F y P (por "obligatorio", "prohibido" y "está permitido") en la lógica deóntica, *F* y *P* (por "en el futuro sucederá que" y "en el pasado sucedió que") en la lógica temporal, K (por "el agente sabe que") en la lógica epistémica, <π> y [π] (por "alguna ejecución finita del programa π" y "toda ejecución finita del programa π") en la lógica dinámica.

Ejemplos de expresiones de distintas lógicas modales:

F(p)	prohibido pisar el césped	*lógica deóntica*
P(p)	en el pasado pisé el césped	*lógica temporal*
<π>q	alguna ejecución de π arroja información q	*lógica dinámica*

Las interpretaciones y usos actuales de la lógica modal caen dentro de dos grandes áreas: la de la *información* y la de la *acción*.

Nota. Los principios generales. Pareciera que un principio general de la lógica modal es □p → ◊p, cuya lectura intuitiva es "lo que es necesario, es posible". Sin embargo, a pesar de

que dicha fórmula luce consistente desde el sentido común, no es un principio rector de la lógica modal (lo justificamos más adelante, al manejar el aparato formal). Considerar como principios generales de la lógica modal a diferentes fórmulas es difícil de decidir. También es difícil determinar qué fórmulas merecen ser consideradas principios de una lógica modal de propósito determinado como lo es la lógica modal temporal, que es una lógica modal para representar el tiempo, o la lógica deóntica, que es una lógica modal para formalizar normas.

Discusión. En la *lógica epistémica*, que se ocupa de precisar aspectos referidos al *conocimiento*, usamos Kp para simbolizar "el agente sabe p". Las fórmulas $Kp \rightarrow p$, $p \rightarrow Kp$, y $Kp \rightarrow KKp$, ¿podrían ser consideradas principios rectores de la lógica epistémica? ¿Cuál es la lectura intuitiva de cada una de ellas?

De aquí al final del capítulo nos concentramos en lenguajes modales con una, o a lo sumo dos modalidades, con o sin sus duales, y de aridad 1 (la aridad de un operador es la cantidad de argumentos para que el operador pueda funcionar). Pero no siempre debemos restringirnos así; existen lógicas modales con infinitos operadores (la lógica dinámica, por ejemplo), y lógicas modales con operadores de aridad mayor a 1.

Usamos también el concepto usual de sustitución uniforme, que permite reemplazar en una fórmula todas las apariciones de una subfórmula por otra. Entonces, por ejemplo, dada la fórmula $p \wedge q \wedge r$, y dada la sustitución $\sigma = \{p/(p \wedge \Box q), q/(\Diamond\Diamond q \vee r)\}$ tenemos que $[p \wedge q \wedge r]_\sigma = (p \wedge \Box q) \wedge (\Diamond\Diamond q \vee r) \wedge r$.

Semántica. Asociado a un lenguaje modal hay estructuras matemáticas en las que definimos las nociones de consecuencia lógica y verdad. Pasamos entonces a ver estas estructuras: frame y modelo.

Un *frame* es una dupla $\mathcal{F} = (W,R)$ tal que W es un conjunto no vacío llamado universo (o dominio) de $\mathcal{F}$, y R es una relación binaria sobre W. Los elementos en W se llaman *puntos*, *situaciones*, *estados*, o *mundos*, y a R se la denomina *relación de accesibilidad* entre mundos. Por ejemplo, el frame formado por los números naturales con la relación "<", $\mathcal{F} = (N,<)$, es un frame usual de la lógica temporal en el que podemos interpretar a cada mundo como un día, o una hora, o una semana. El frame formado por los números reales con la relación "<", $\mathcal{F} = (R,<)$ es también un frame usual para interpretar el tiempo y nos permite considerar al tiempo como *denso*: si cada mundo se corresponde con un número real que representa un instante de tiempo, entonces es posible identificar otro instante de tiempo entre cada par de instantes. Notar que en estos frames asumimos que tanto el pasado como el futuro son una *línea temporal*, pero tengamos en cuenta que existen concepciones del tiempo no determinísticas, donde el futuro y/o el pasado tienen una estructura de árbol.

Dado un lenguaje modal, un *modelo* es un par $\mathcal{M} = (\mathcal{F},V)$ donde $\mathcal{F} = (W,R)$ es un frame y $V: P \rightarrow \mathcal{P}(W)$ es una *función de valuación* que asigna a cada proposición p del lenguaje un subconjunto V(p) de W. Intuitivamente, V(p) es el conjunto de mundos en los que vale p. Los

modelos, así presentados, son frames a los que les agregamos una función de valuación. A estos modelos se los llama *modelos de Kripke*.

Así como evaluamos fórmulas de la lógica proposicional en el conjunto de valores de verdad booleanos representado por las constantes en el conjunto {*true*, *false*}, y así como evaluamos fórmulas de la lógica de predicados en una estructura que llamamos interpretación (que por definición consta de un conjunto no vacío de elementos llamado dominio, una colección de elementos distinguidos llamados constantes, una colección de funciones sobre elementos del dominio y una colección de relaciones sobre elementos del dominio), en la lógica modal evaluamos fórmulas en modelos (y también en frames). Más adelante en este capítulo veremos que las estructuras de las interpretaciones de la lógica de predicados y las estructuras de los frames y modelos de la lógica modal guardan, en realidad, una muy estrecha relación entre sí.

Clásicamente tenemos la siguiente definición inductiva de cuándo una fórmula es verdadera en un modelo $\mathcal{M} = (\mathcal{F},V)$ en un mundo *w*. Recordemos que $\mathcal{A} \models \mathcal{B}$ se lee coloquialmente "de $\mathcal{A}$ se deduce $\mathcal{B}$", o "$\mathcal{B}$ es consecuencia lógica de $\mathcal{A}$". En la definición que sigue, $\mathcal{M},w \models \mathcal{A}$ se lee coloquialmente "$\mathcal{A}$ es localmente verdadera en un mundo *w* en un modelo $\mathcal{M}$".

Definición 3.2. Verdad local. Sea p $\in$ *P*, sean $\mathcal{A}, \mathcal{B} \in \mathcal{L}$:

$\mathcal{M},w \models p$ si y solo si $w \in V(p)$

Nunca sucede que $\mathcal{M},w \models \bot$

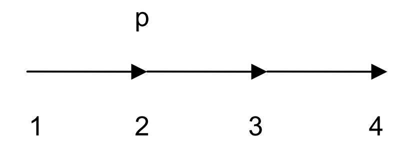
$\mathcal{M},w \models \neg\mathcal{A}$ si y solo si no sucede que $\mathcal{M},w \models \mathcal{A}$

$\mathcal{M},w \models \mathcal{A} \vee \mathcal{B}$ si y solo si $\mathcal{M},w \models \mathcal{A}$ o $\mathcal{M},w \models \mathcal{B}$

$\mathcal{M},w \models \Diamond\mathcal{A}$ si y solo si existe un mundo *v* tal que R*wv* y $\mathcal{M},v \models \mathcal{A}$

Algunos comentarios. i) La segunda condición en la definición 3.2 establece que en un mundo cualquiera no puede valer una contradicción (que es distinto a que una propiedad p valga o no valga en un mundo cualquiera); ello tiene sentido pues es de esperar que las contradicciones no valgan nunca en ninguna parte. ii) La idea de "posibilidad" surge naturalmente cuando notamos que para que una fórmula $\Diamond\mathcal{A}$ sea verdadera en *w* se requiere que $\mathcal{A}$ sea verdadera en algún mundo *v* R-accesible desde *w*. iii) Dado que consideramos al símbolo $\Box$ una abreviatura de "$\neg\Diamond\neg$", de la última condición en la definición 3.2 surge que $\mathcal{M},w \models \Box\mathcal{A}$ si y solo si para todo mundo *v* tal que R*wv* sucede que $\mathcal{M},v \models \mathcal{A}$. iv) Finalmente, notemos que la noción de verdad dada es "local" a un mundo *w* de un modelo $\mathcal{M}$ para un frame $\mathcal{F}$. Esta definición puede asimilarse al concepto de satisfactibilidad que conocemos de la lógica proposicional y la lógica de predicados.

Ejemplo. En el siguiente frame la proposición p es localmente verdadera en el mundo 2, es falsa en los mundos 3 y 4, y la proposición $\Diamond$p es localmente verdadera en 1.



Nota. El frame bidireccional para la lógica temporal. Definimos la estructura de un modelo básico para la lógica temporal $\mathcal{M}$ = (T,R,V), y la semántica de sus operadores *F* y *P* como:

$\mathcal{M},t \models F\mathcal{A}$ si y solo existe un mundo *s* tal que R*ts* y $\mathcal{M},s \models \mathcal{A}$, y con

$\mathcal{M},t \models P\mathcal{A}$ si y solo existe un mundo *s* tal que R*st* y $\mathcal{M},s \models \mathcal{A}$.

Esta definición respeta la definición 3.2; a su vez, notemos cómo el operador *F* va "hacia adelante" en R y el operador *P* va "hacia atrás" en R, logrando el movimiento intuitivo pretendido en la línea del tiempo.

Nota. Evaluación de fórmulas. Sea $\mathcal{F}$ = (W,R) un frame y sea $w \in$ W un mundo en un modelo $\mathcal{M}$ = ($\mathcal{F}$,V). Extendemos naturalmente la función de valuación V, en el sentido inductivo usual, para evaluar fórmulas: V($\mathcal{A}$) = $\{w / \mathcal{M},w \models \mathcal{A}\}$.

En el gráfico previo, la proposición p $\vee$ q es localmente verdadera en el mundo 2, y la proposición $\Box$p $\wedge$ $\Diamond$p es localmente verdadera en el mundo 1.

Además de querer saber si una fórmula es localmente verdadera o no, podemos querer saber si una fórmula es *globalmente verdadera*, esto es, si es verdadera en todos los mundos de un modelo dado. O si no lo es, claro.

Definición 3.3. Verdad global. Sea $\mathcal{A}$ una fórmula, sea $\mathcal{M}$ = ($\mathcal{F}$,V) un modelo. $\mathcal{A}$ es globalmente verdadera en $\mathcal{M}$, escribimos $\mathcal{M} \models \mathcal{A}$, si $\mathcal{A}$ es localmente verdadera en todos los mundos de W en $\mathcal{M}$.

En el gráfico del ejemplo previo es fácil ver que la fórmula $\neg$q es globalmente verdadera en el modelo.

Sabemos que si a un frame le agregamos información contingente (una valuación) tenemos un modelo. Pero podemos querer ignorar la información contingente (la que nos dice qué fórmula vale en qué mundo) y averiguar qué fórmulas son verdaderas con respecto a la estructura del frame. Esto es, podemos "olvidarnos" de la información contingente –de todos los modelos que existen para un frame– y averiguar qué información es verdadera respecto de la estructura del frame. Esta es una noción de verdad, si se quiere, "más fuerte" pues se enfoca en la estructura (más "sólida", "estática", o "fija") y no en las valuaciones (más "dinámicas", o "volátiles" que pueden suceder o no suceder):

Definición 3.4. Validez. Sean $\mathcal{A}$ una fórmula, $\mathcal{F}$ = (W,R) un frame, $w \in$ W un mundo:

a) $\mathcal{A}$ es *válida en un mundo w* en un frame $\mathcal{F}$ ($\mathcal{F},w \models \mathcal{A}$) si $\mathcal{M},w \models \mathcal{A}$ para todo modelo $\mathcal{M}$ = ($\mathcal{F}$,V); es decir, cuando $\mathcal{A}$ es localmente verdadera en w para cualquier modelo $\mathcal{M}$ "basado" en $\mathcal{F}$.

b) $\mathcal{A}$ es *válida en un frame* $\mathcal{F}$ ($\mathcal{F} \models \mathcal{A}$) si $\mathcal{A}$ es válida en todo mundo w en $\mathcal{F}$ = (W,R).

c) $\mathcal{A}$ es *válida en una clase de frames* F (F $\models \mathcal{A}$) si $\mathcal{A}$ es válida en todo frame $\mathcal{F} \in$ F.

d) $\mathcal{A}$ es *válida* ($\models \mathcal{A}$) si $\mathcal{A}$ es válida en **F**, la clase de todos los frames.

Ejemplo. La fórmula $\Diamond\Diamond p \rightarrow \Diamond p$ es válida en la clase de los frames transitivos. Para comprobar esta afirmación, sea $\mathcal{F}$ = (W,R) un frame transitivo cualquiera (esto es, $\mathcal{F}$ verifica la propiedad $\forall xyz$ (R$xy \wedge$ Ryz) $\rightarrow$ (Rxz), con x, y, $z \in$ W), y sea w cualquier mundo de W. Si sucede que $\Diamond\Diamond p$ es verdadera en w entonces existe un v tal que Rwv y $\mathcal{F},v \models \Diamond p$; y si esto ocurre entonces existe un u tal que Rvu y $\mathcal{F},u \models$ p. Como $\mathcal{F}$ es transitivo tenemos que vale Rwu, con lo que $\mathcal{F},w \models \Diamond p$. Por lo tanto la fórmula $\Diamond\Diamond p \rightarrow \Diamond p$ es válida en cualquier w en cualquier $\mathcal{F}$ transitivo y, por definición, es válida en la clase de los frames transitivos.

Discusión. ¿Cuáles son las fórmulas a las que hace referencia la definición 3.4.d?

Recordemos que, si bien desde un punto de vista filosófico podemos decir que existe una única "noción de verdad" que todos aspiramos a conocer, cuando manipulamos un sistema formal como la lógica modal –que intenta capturar y usar la noción de verdad (y la de falsedad)– solo podemos acercarnos a ésta a través de las herramientas que el propio sistema formal nos provee. Con lo cual el camino hacia la verdad siempre se nos presenta relativo a la herramienta que usamos para llegar a ella, y por lo tanto, la noción de verdad se vuelve de algún modo *relativa al sistema formal* que usamos. Para ilustrar este punto, pensemos en que la lógica proposicional maneja los conceptos de verdad y falsedad de un modo simple y llano armando las tablas de verdad de acuerdo a las funciones de verdad de los conectivos. Luego, cuando usamos la lógica de predicados (que es más rica que la lógica proposicional) aparecen nuevas estructuras sobre las que testear validez y con éstas aparecen las nociones de verdad en una interpretación y validez lógica (verdad en todas las interpretaciones). Al trabajar con la lógica modal vemos que sucede lo mismo: tenemos distintas maneras de acceder al concepto de verdad, maneras sensibles a las estructuras con las que trabaja la lógica en cuestión.

Relaciones de consecuencia lógica. Tenemos ya formada una intuición del concepto de consecuencia lógica, y es la que dice que la validez de las premisas garantiza la validez de la conclusión. Hemos visto esta intuición formalizada en los capítulos 1 y 2, al estudiar la lógica proposicional y la lógica de predicados.

En la lógica modal las consecuencias lógicas dependerán de la estructura con la que estemos trabajando. Esto quiere decir que la noción de consecuencia lógica está *parametrizada*. Definimos a

continuación, semánticamente, las nociones de *consecuencia local* y *consecuencia global*. La noción de consecuencia local es la noción de consecuencia lógica que ya hemos manejado en la lógica proposicional y la lógica de predicados, trasladada a la lógica modal.

Definición 3.5. Consecuencia lógica "local". Sea Σ un conjunto de fórmulas, sea $\mathcal{A}$ una fórmula, sea S una clase de estructuras (modelos, frames,…). Decimos que $\mathcal{A}$ es consecuencia local de Σ sobre S, $\Sigma \models_S \mathcal{A}$, si para todos los modelos $\mathcal{M}$ de S (si S son modelos, para ellos mismos, si S son frames, para todos los modelos de ellos) y todos los mundos *w* de $\mathcal{M}$, si sucede que $\mathcal{M},w \models \Sigma$ entonces $\mathcal{M},w \models \mathcal{A}$.

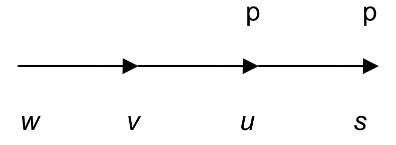
Ejemplo. Es fácil ver que $\{\Diamond\Diamond p\} \models_{\mathbf{Tran}} \Diamond p$, con Tran la clase de los frames transitivos. Pero también es directo notar que $\Diamond p$ no es una consecuencia local de $\{\Diamond\Diamond p\}$ en la clase **F** de todos los frames (para comprobarlo, proveer un *contramodelo*).

Definición 3.6. Consecuencia lógica "global". Sea Σ un conjunto de fórmulas, sea $\mathcal{A}$ una fórmula, sea S una clase de estructuras (modelos, frames,…). Decimos que $\mathcal{A}$ es una consecuencia global de Σ sobre S, $\Sigma \models^g_S \mathcal{A}$, si para toda estructura *S* en S, si *S* $\models \Sigma$ entonces *S* $\models \mathcal{A}$. Aquí, dependiendo del tipo de estructuras que contiene S, el símbolo "$\models$" se interpreta como validez en un frame (si S es una clase de frames), verdad global (en un modelo, si S es un conjunto de modelos), etc.

Ejemplo. Vale $\{\Diamond\Diamond\Diamond p \rightarrow \Diamond\Diamond p\} \models^g_{\mathbf{F}} \Diamond\Diamond p \rightarrow \Diamond p$; pero no vale $\{\Diamond\Diamond\Diamond p \rightarrow \Diamond\Diamond p\} \models_{\mathbf{F}} \Diamond\Diamond p \rightarrow \Diamond p$, siendo **F** la clase de todos los frames.

¿Por qué la primera afirmación es cierta? Porque las fórmulas $\Diamond\Diamond\Diamond p \rightarrow \Diamond\Diamond p$ y $\Diamond\Diamond p \rightarrow \Diamond p$ valen en los frames transitivos (probarlo); por lo tanto vale la consecuencia lógica global en la clase **F** de todos los frames (notar que en aquellos frames donde la subfórmula $\{\Diamond\Diamond\Diamond p \rightarrow \Diamond\Diamond p\}$ es falsa, la afirmación es verdadera).

Ahora bien, no vale la afirmación $\{\Diamond\Diamond\Diamond p \rightarrow \Diamond\Diamond p\} \models_{\mathbf{F}} \Diamond\Diamond p \rightarrow \Diamond p$. Esto es, la fórmula $\Diamond\Diamond p \rightarrow \Diamond p$ no es consecuencia lógica local de la fórmula $\Diamond\Diamond\Diamond p \rightarrow \Diamond\Diamond p$ teniendo en cuenta la clase de todos los frames. Ello porque podemos construir el "contramodelo" $\mathcal{M}$ con la siguiente estructura finita:



y con V(p) = $\{u, s\}$. Entonces tenemos que: $\mathcal{M},w \models \Diamond\Diamond\Diamond p$, y $\mathcal{M},w \models \Diamond\Diamond p$, y por lo tanto $\mathcal{M},w \models \Diamond\Diamond\Diamond p \rightarrow \Diamond\Diamond p$. Y si bien $\mathcal{M},w \models \Diamond\Diamond p$, no es cierto que $\mathcal{M},w \models \Diamond p$ (pues, para que ello sucediera, deberíamos tener $\mathcal{M},v \models p$). Con lo que no vale $\mathcal{M},w \models \Diamond\Diamond p \rightarrow \Diamond p$.

Vemos a continuación algunas definiciones y herramientas sintácticas que permiten manejar las relaciones semánticas de validez y consecuencia lógica de un modo más automatizado. Esto es importante para nosotros como informáticos.

Definición 3.7.a. Lógica modal. Una lógica modal Λ es un conjunto de fórmulas bien formadas que contiene todas las tautologías proposicionales, es cerrado –está *clausurado*– bajo *modus ponens* (esto es, si las fórmulas p y p $\rightarrow$ q pertenecen a Λ, entonces la fórmula q también), y es cerrado bajo sustitución uniforme (si una fórmula $\mathcal{A}$ pertenece a Λ entonces todas sus instancias de sustituciones también).

Si una fórmula $\mathcal{A}$ pertenece a Λ decimos que $\mathcal{A}$ es *teorema* de Λ. Si Λ_1 y Λ_2 son dos lógicas modales y $\Lambda_1 \subseteq \Lambda_2$ decimos que Λ_2 es una *extensión* de Λ_1.

Definición 3.7.b. Lógica modal normal. Una lógica modal Λ es *normal* si contiene las fórmulas $\Box(p \rightarrow q) \rightarrow (\Box p \rightarrow \Box q)$ y $\neg\Box\neg p \leftrightarrow \Diamond p$, y es cerrada bajo *generalización* (esto es, si una fórmula $\mathcal{A}$ pertenece a Λ, entonces $\Box\mathcal{A}$ también).

Observemos que estas dos definiciones son bien simples: identifican a una lógica como un conjunto de fórmulas que cumplen ciertas condiciones de clausura.

De la definición 3.7.a se desprende que la lógica proposicional –tal como la estudiamos en el primer capítulo– está contenida en una lógica modal. De ambas definiciones podemos intuir que existen lógicas no normales, como veremos más adelante. Dicho de modo simple, la "normalidad" de una lógica modal queda determinada por la propiedad de distribución del "$\Box$" sobre el "$\rightarrow$" y por la regla de generalización.

Damos a continuación la definición sintáctica (axiomática) de una lógica modal.

Definición 3.8. Sistema formal K de la lógica modal.

Lenguaje

$\mathcal{L}$ (como en la definición 3.1).

Axiomas

Todas las instancias de tautologías proposicionales.

(K) $\Box(p \rightarrow q) \rightarrow (\Box p \rightarrow \Box q)$.

(Dual) $\neg\Box\neg p \leftrightarrow \Diamond p$.

Reglas de inferencia

Modus ponens: a partir de p y de p $\rightarrow$ q obtenemos q.

Sustitución uniforme: a partir de una fórmula $\mathcal{A}$ conseguimos una fórmula $\mathcal{B}$ sustituyendo uniformemente letras proposicionales en $\mathcal{A}$ por fórmulas arbitrarias.

Generalización: si tenemos p obtenemos $\Box$p.

Algunos comentarios y observaciones sobre el sistema formal K. Traigamos a este punto la noción de "derivación" (o "deducción" o "demostración") que ya conocemos (lo hemos visto en el estudio de la lógica proposicional y la lógica de predicados). Sabemos que una *derivación* de $\mathcal{A}$ a partir de un conjunto finito Γ de fórmulas bien formadas es una secuencia finita de fórmulas bien formadas $\mathcal{A}_1,..., \mathcal{A}_n$ en la que $\mathcal{A}_n = \mathcal{A}$ y, para todo *i*, cada $\mathcal{A}_i$ de la secuencia es: o una instancia de uno de los esquemas de axioma provistos por el sistema formal, o es una fórmula en Γ, o se obtiene por aplicación de la regla de *modus ponens* entre dos fórmulas $\mathcal{A}_k$ y $\mathcal{A}_j$ que aparecen antes en la secuencia (esto es, con $k,j < i$), o se obtiene de la aplicación directa de la regla de generalización sobre alguna fórmula $\mathcal{A}_k$, con $k \leq i$. Esta última condición es la única condición "novedosa" que aparece ahora para la noción de derivación relativa a este sistema formal de la lógica modal. Cuando Γ es el conjunto vacío, entonces decimos que $\mathcal{A}$ es *teorema* de K. Para simbolizar que $\mathcal{A}$ es derivable a partir de Γ en K escribimos $\Gamma \vdash_K \mathcal{A}$. Notemos que la noción de derivación nos permite pensar que el sistema formal K dado en la definición 3.8 induce una lógica modal en el sentido de la definición 3.7.a, y que el conjunto de las fórmulas derivables es una lógica en el sentido de la definición 3.7.b.

Otras observaciones. El sistema modal K dado en la definición 3.8 es el mínimo normal, esto es, es el sistema modal normal que tiene menos restricciones. Vemos que, aplicando la regla de sustitución, podemos construir nuevas tautologías provenientes de la lógica proposicional y que ahora contengan las modalidades $\Box$ y $\Diamond$ (por ejemplo: $\neg\Diamond p \vee \Diamond p$). Es fácil ver que estas tautologías son válidas en todos los frames (queda para el lector la tarea de demostrar esta afirmación). Otras fórmulas válidas en todos los frames –como las que, por ejemplo, podemos obtener a partir del axioma K en un solo paso– no provienen por sustitución de ninguna tautología proposicional, pues la lógica proposicional carece del símbolo "$\Box$". *Modus ponens* preserva validez en frames, verdad global y verdad local (dejamos al lector estas pruebas). Sustitución uniforme no preserva ni verdad global ni satisfactibilidad en un mundo (q se obtiene por sustitución de p, pero si p es verdad global en un modelo, no necesariamente lo es q. Probarlo construyendo un modelo apropiado).

Decimos que el axioma K permite realizar razonamiento proposicional tradicional porque el "$\Box$" se "mete" dentro del paréntesis y se "distribuye": entonces pasamos de tener una fórmula modal a tener un condicional entre dos subfórmulas modales. La regla de generalización permite crear nuevas fórmulas modales yuxtaponiendo un "$\Box$" delante de una fórmula demostrable. Con K y generalización tenemos entonces, de algún modo, un "interjuego" entre dos contextos: el proposicional y el modal. Generalización preserva verdad global (si p vale para todo mundo en un modelo entonces vale $\Box p$ porque en cada mundo vale p para todos sus adyacentes) pero no preserva satisfactibilidad (si p es verdadera en un mundo no podemos afirmar que en el mundo vale $\Box p$).

Extensiones de K. El sistema modal K es un sistema formal mínimo y simple. Dado cualquier conjunto Γ de fórmulas modales, podemos agregarlas como nuevos axiomas y formar el sistema modal KΓ. Esta técnica generadora de nuevos sistemas formales es sintáctica y la hemos usado previamente para generar extensiones.

Definir una lógica estableciendo las fórmulas que genera –esto es, agregar axiomas que para nosotros tienen algún tipo de interés– es un modo usualmente aceptado de especificar lógicas. Sin embargo, pareciera haber algo arbitrario en este proceso de definir lógicas "sintácticamente": ¿por qué agregaríamos algunas fórmulas como punto de partida de nuevos teoremas, y no otras?

Es útil también –desde el punto de vista formal– conocer cuál es la contraparte semántica de una lógica.

En muchos casos es posible describir a las extensiones del sistema K en términos de validez en frames. Esta es una perspectiva *semántica*. Con la siguiente definición tenemos entonces una manera diferente de especificar una lógica modal a como lo hemos hecho en la definición 3.7.a. Describimos *semánticamente* una lógica modal mediante la identificación de la estructura de los frames en los que son válidas las fórmulas de la lógica en cuestión.

Definición 3.9. Lógica modal (desde una perspectiva semántica). Sea S una clase de frames. Definimos el conjunto de fórmulas $\Lambda_S = \{\mathcal{A} / S \models \mathcal{A}$, para todas las estructuras $S \in$ S$\}$, con $\mathcal{A}$ fórmulas del lenguaje modal; Λ_S es una lógica modal.

La relación entre el aspecto sintáctico y el aspecto semántico de las lógicas modales nos lleva a considerar resultados de *correctitud* (o *sensatez* como usamos en los capítulos anteriores) y *completitud* de dichas lógicas. Intuitivamente sabemos que la relación entre sintaxis y semántica debe ser tal que los teoremas que derivamos en la lógica son verdaderos y que todas las fórmulas verdaderas tienen una derivación sintáctica de la cual dicha fórmula es el último paso.

Existen teoremas que relacionan a las lógicas modales descriptas axiomáticamente con las estructuras de los frames en los que sus teoremas son válidos: estos teoremas a veces se conocen como *teoremas de determinación.*

Ejemplos. El axioma $\Diamond\Diamond p \rightarrow \Diamond p$ identifica a la lógica modal cuyas fórmulas son verdaderas en los frames transitivos, tal como hemos visto en dos ejemplos estudiados previamente en este capítulo. El axioma $p \rightarrow \Diamond p$ identifica a la lógica modal cuyas fórmulas son verdaderas en los frames reflexivos. El axioma $\Box p \rightarrow \Diamond p$ identifica a las lógicas modales cuyas fórmulas son verdaderas en los frames sin límite a derecha (*right-unbounded*). Las respectivas extensiones del sistema K para estas tres lógicas se llaman K4, KT, y KD. La clase de frames cuya relación es una relación de equivalencia (esto es, verifica reflexividad, transitividad y simetría) se identifican con la extensión de K conocida como S5.

A continuación damos las definiciones de correctitud y completitud de una lógica modal.

Definición 3.10. Correctitud (o sensatez). Sea S una clase de estructuras (concentrémonos en frames). Una lógica modal normal Λ es correcta (o sensata) con respecto a S si $\Lambda \subseteq \Lambda_S$, con $\Lambda_S = \{\mathcal{A} \,/\, S \models \mathcal{A} \;\; \forall\, S \in \mathrm{S}\}$.

Notemos que esta definición es por inclusión de un conjunto de fórmulas en otro conjunto (ya hemos usado este estilo de definición en la definición 3.7.a).

De modo equivalente puede definirse que la lógica Λ es correcta con respecto a S si para toda fórmula $\mathcal{A}$ y todas las estructuras $S \in \mathrm{S}$, $\vdash_\Lambda \mathcal{A}$ implica $S \models \mathcal{A}$. Esto es, si $\mathcal{A}$ es teorema en Λ entonces es válida en *S*. Decimos entonces que S *es una clase de estructuras para* Λ.

Prueba de correctitud. Para probar la correctitud de una lógica modal normal (presentada en términos de axiomas y reglas de inferencia) respecto de una clase de frames debemos probar que los axiomas de la lógica son válidos en la clase de frames de que se trate y que las reglas de inferencia (*modus ponens*, generalización y sustitución uniforme) preservan verdad.

A continuación damos la definición de completitud "fuerte" (existe también una definición de completitud "débil"). Que un sistema formal sea completo, genéricamente hablando, significa que *lo que es cierto en el sistema entonces es demostrable en el sistema*.

Definición 3.11. Completitud (fuerte). Sea S una clase de frames. Una lógica modal normal Λ es "fuertemente" completa con respecto a S si, para cualquier conjunto de fórmulas $\Gamma \cup \{\mathcal{A}\}$, si $\Gamma \models_S \mathcal{A}$ entonces $\Gamma \vdash_\Lambda \mathcal{A}$.

Teorema de completitud de K. El sistema formal K de la lógica modal es fuertemente completo respecto de la clase de todos los frames.

Los teoremas de completitud son, esencialmente, teoremas de existencia de modelos. Esto es, para probar completitud usualmente hay que probar que determinados modelos "especiales" existen. Lo importante entonces es que sepamos cómo encontrarlos o cómo construirlos.

Para demostrar el teorema de completitud de K necesitamos conocer algunas definiciones y hacer algunos comentarios previos:

a) Un conjunto Γ de fórmulas es *consistente* si, o bien $\mathcal{A}$ o bien $\neg\mathcal{A}$ no es teorema de Γ (ambas no son teoremas a la vez en Γ). Pensemos que si $\mathcal{A}$ y $\neg\mathcal{A}$ son teoremas de Γ entonces de Γ se deduce una contradicción ($\mathcal{A} \wedge \neg\mathcal{A}$) que podemos usarla como premisa para derivar todas las fórmulas del lenguaje. De un conjunto inconsistente pueden derivarse todas las fórmulas. La consistencia es una característica importante de los conjuntos de fórmulas: poco interés tiene –tanto desde el punto de vista lógico como desde

el punto de vista de los sistemas informáticos– un conjunto de fórmulas a partir del cual pueden derivarse todas las demás.

b) Un conjunto Γ de fórmulas de una lógica Λ es *maximal Λ-consistente* si es consistente y cualquier otro conjunto Δ de fórmulas tal que $\Gamma \subset \Delta$ es Λ-inconsistente.

c) La propiedad de *compacidad* (por *compactness* en inglés) establece que un conjunto Γ de fórmulas de una lógica Λ es Λ-consistente si y solo si todo subconjunto finito de Γ lo es. Daremos un esquema de la demostración de esta propiedad hacia el final de esta sección.

La técnica de prueba que se usa para demostrar el teorema de completitud de K se conoce como de *completitud por canonicidad*: se construyen modelos, llamados *canónicos*, a partir de conjuntos maximales consistentes.

Notemos dos detalles relevantes. Por un lado, todo mundo *w* en todo modelo $\mathcal{M}$ para una lógica Λ está asociado con el conjunto de fórmulas $\{\mathcal{A} \,/\, \mathcal{M},w \models \mathcal{A}\}$, esto es, el conjunto de fórmulas que son verdaderas en *w*. Es fácil verificar que este conjunto de fórmulas es maximal Λ-consistente (es decir, si $\mathcal{A}$ es verdadera en algún modelo para Λ entonces $\mathcal{A}$ pertenece a un conjunto maximal Λ-consistente). Por otro lado, si el mundo *w* está relacionado con otro mundo *v* en un modelo $\mathcal{M}$ debe quedarnos claro que la información codificada en los conjuntos maximales Λ-consistentes de *w* y de *v* está relacionada, digamos, “de algún modo coherente”. Podemos entonces formarnos la intuición de que los modelos permiten que conjuntos maximales consistentes se relacionen coherentemente entre sí.

La idea detrás de la construcción de modelos canónicos es poner a trabajar estos dos detalles relevantes recién señalados: partir de colecciones de conjuntos maximales consistentes “coherentemente relacionados” e intentar obtener el modelo buscado. El objetivo es probar que la afirmación “$\mathcal{A}$ pertenece a un conjunto maximal Λ-consistente” es equivalente a “$\mathcal{A}$ es verdadera en algún modelo” (esta afirmación es un *lema de verdad*). Se prueba construyendo un modelo especial - el modelo canónico - cuyos mundos son todos los conjuntos maximales consistentes de la lógica Λ. Veamos la definición siguiente:

Definición 3.12. Modelo canónico. El modelo canónico para una lógica modal normal Λ es la terna $(W^\Lambda,R^\Lambda,V^\Lambda)$ con:

- W^Λ , el conjunto de todos los conjuntos maximales Λ-consistentes.
- R^Λ , la relación canónica sobre W^Λ, definida como: $R^\Lambda wv$ si para toda fórmula $\mathcal{A}$, $\mathcal{A} \in v$ implica $\Diamond\mathcal{A} \in w$.
- V^Λ, la función de valuación canónica, definida como $V^\Lambda(p) = \{w\in W^\Lambda / p\in w\}$.

W^Λ contiene todos los conjuntos maximales Λ-consistentes. Esto es relevante porque (por el *lema de Lindenbaum*) cualquier conjunto Λ-consistente de fórmulas es un subconjunto de algún mundo de W^Λ y entonces (por el lema de verdad) cualquier conjunto Λ-consistente de fórmulas es verdadero en algún mundo del modelo.

R^{Λ} es una relación de accesibilidad entre conjuntos maximales consistentes basada (precisamente) en el concepto de consistencia. Como los mundos en W^{Λ} son conjuntos maximales consistentes, si en el mundo *v* adyacente a *w* la fórmula $\mathcal{A}$ no fuese cierta entonces en *v* valdría $\neg\mathcal{A}$ pero entonces por definición de R^{Λ} en *w* valdría $\Diamond\neg\mathcal{A}$, lo que es absurdo por ser *w* un conjunto consistente.

Finalmente, la función de valuación canónica V^{Λ} iguala la verdad de un símbolo proposicional en *w* con su pertenencia a *w*. Así, el modelo canónico nos permite relacionar verdad con pertenencia a un conjunto maximal consistente.

Ya estamos en condiciones de organizar un esquema de la prueba del teorema de completitud fuerte de K.

Orientación para la prueba del teorema de completitud fuerte de K. Para probar la completitud fuerte de K hay que usar la noción de compacidad. Tenemos que encontrar, para cada conjunto Γ K-consistente de fórmulas, un modelo $\mathcal{M}$ y un mundo *w* en $\mathcal{M}$ tal que $\mathcal{M},w \models \Gamma$. Elegimos $\mathcal{M} = (\mathcal{F}^K,V^K)$, el modelo canónico para K, y elegimos que el mundo *w* sea cualquier conjunto maximal consistente Γ^+ que extienda a Γ. Entonces $(\mathcal{F}^K,V^K),\ \Gamma^+ \models \Gamma$. Ciertamente podemos elegir a $\mathcal{M} = (\mathcal{F}^K,V^K)$ porque un resultado auxiliar (y relevante) nos lo garantiza: el lema de Lindenbaum asegura que si Γ es un conjunto Λ-consistente de fórmulas, entonces Γ^+ existe.

Finalmente, para terminar esta presentación de la noción de completitud, mencionamos que existe aún un resultado más poderoso y general llamado *teorema del modelo canónico* que afirma que toda lógica modal normal es fuertemente completa respecto de su modelo canónico. Su demostración se apoya en el lema de Lindenbaum y en la técnica de armado de modelos canónicos vista.

Computabilidad y complejidad de las lógicas modales. Para los informáticos es importante conocer aspectos de computabilidad y complejidad de las lógicas modales. Esto significa conocer cuántos recursos de tiempo (pasos de computación) y de espacio (memoria) se necesitan para saber si una fórmula es satisfactible en un modelo de una lógica dada.

Decidibilidad. Sabemos que un conjunto Γ de fórmulas es decidible si existe un procedimiento (un método finito y efectivo de decisión) para determinar si cualquier fórmula del lenguaje pertenece a Γ.

Decimos entonces que una lógica modal normal Λ es decidible si el problema de *Λ-satisfactibilidad* (determinar si una fórmula $\mathcal{A}$ es satisfactible en algún modelo para Λ) es decidible.

Existe otro problema interesante referido a decidibilidad de las lógicas modales y que se basa en el problema de Λ-satisfactibilidad: es el problema de *Λ-validez*, que consiste en determinar si una fórmula $\mathcal{A}$ es válida en la clase de modelos M que identifica a la lógica Λ.

A continuación presentamos informalmente el problema de cómo se establecen resultados de Λ-*satisfactibilidad* y Λ-*validez* para una lógica modal.

Hemos visto que podemos tener una lógica modal especificada de manera puramente semántica, conociendo la clase de frames que la identifican. Y que también podemos conocerla desde su aspecto puramente sintáctico, sabiendo cuáles son los axiomas y las reglas que generan la lógica. También sabemos que la computación trata la manipulación finita de estructuras finitas.

Sin importarnos si la lógica se nos presenta desde su aspecto sintáctico o desde su aspecto semántico, debemos poder determinar si es decidible o no. Un instrumento para demostrar la decidibilidad de una lógica es el siguiente:

Propiedad de modelo finito (f.m.p. por *finite model property en inglés*). Sea Λ una lógica, y M una clase de modelos para Λ. Decimos que Λ tiene la propiedad de modelo finito con respecto a M si dada una fórmula $\mathcal{A}$ de Λ que es satisfactible en algún modelo en M entonces $\mathcal{A}$ es satisfactible en un modelo finito en M. Un modelo es finito si su conjunto de mundos W tiene una cantidad finita de elementos, si no el modelo es infinito.

La f.m.p. es interesante para nosotros como informáticos porque es una fuente de robustez computacional de la lógica modal: no tenemos que preocuparnos por un modelo infinito porque si vale la f.m.p. para la lógica en cuestión entonces siempre podemos encontrar otro modelo finito que es, de algún modo, “equivalente” al infinito. No entraremos en detalles de las técnicas de obtención de modelos finitos, pero informalmente mencionaremos dos: *selección* y *filtrado*. La primera elige cuidadosamente un submodelo finito del modelo infinito (por ejemplo, eliminando mundos que son redundantes). La segunda encuentra una estructura finita que se corresponde con el modelo infinito de modo que la estructura infinita puede mapearse en la estructura finita.

Discusión informal de decidibilidad para lógicas especificadas semánticamente. Supongamos que tenemos la lógica Λ especificada semánticamente. Supongamos también que sabemos (o probamos) que Λ verifica una forma “fuerte” de f.m.p., esto es: no solo verifica la f.m.p. respecto de alguna clase de modelos sino que además, para cualquier fórmula $\mathcal{A}$ existe una función computable *f* tal que $f(|\mathcal{A}|)$ es una cota superior del tamaño de los modelos necesarios para satisfacer $\mathcal{A}$ (donde $|\mathcal{A}|$ es la “longitud” de $\mathcal{A}$, que puede estar medida tanto en cantidad de subfórmulas como en letras proposicionales). Entonces: escribimos un programa que recibe a $\mathcal{A}$ como input, genera todos los modelos finitos (de la clase de modelos de que se trata) hasta los del tamaño $f(|\mathcal{A}|)$ y testea satisfactibilidad de $\mathcal{A}$ en estos modelos. Como $\mathcal{A}$ es Λ-satisfactible si y solo si es satisfactible en un modelo de Λ de a lo sumo tamaño $f(|\mathcal{A}|)$, y como el programa que construimos examina todos estos modelos, el programa determina Λ-satisfactibilidad.

Discusión informal de decidibilidad para las lógicas especificadas sintácticamente. Tenemos la lógica Λ especificada mediante sus axiomas, y ya probamos que verifica la f.m.p. para alguna clase de modelos M. Entonces escribimos dos programas: uno que usa la axiomatización de Λ para generar las fórmulas Λ-válidas; otro que genera los modelos finitos en M. Si una fórmula $\mathcal{A}$ dada es Λ-válida, entonces será generada por el primer programa; si no lo es, encontraremos con el segundo programa el modelo finito en el que es falsa.

Ejemplo. La lógica modal mínima K es decidible. Verifica la f.m.p. "fuerte".

La prueba de esta propiedad requiere del armado de un modelo finito. Lo hacemos aplicando la técnica de filtrado: dado un modelo $\mathcal{M}$ = (W,R,V) que satisface una fórmula ϕ en algún mundo, filtramos $\mathcal{M}$ usando el conjunto Σ (cerrado) de todas las subfórmulas de ϕ y obtenemos un modelo finito $\mathcal{M}^f$ que satisface ϕ. Escribimos dicho modelo finito como $\mathcal{M}^f_\Sigma = (W^f,R^f,V^f)$ y lo llamamos "el modelo filtrado de $\mathcal{M}$ a partir de Σ". Se arma así:

- W^f es el conjunto de las clases de equivalencia de los mundos de W. Para definir estas clases de equivalencia usamos la siguiente relación de equivalencia:

  $w \cong_\Sigma v$ si y solo si para toda ϕ en Σ ocurre que ($\mathcal{M},w \models \phi$ si y solo si $\mathcal{M},v \models \phi$)

Dos mundos w y v son $\cong_\Sigma$-equivalentes si y solo si ocurre que para toda subfórmula ϕ, ϕ es verdadera en w si y solo si es verdadera en v. Escribimos $|w|_\Sigma$ para referirnos a la clase de equivalencia de w en $\mathcal{M}$ con respecto a $\cong_\Sigma$. El mapeo $w \rightarrow |w|_\Sigma$ que envía cada mundo w a su clase de equivalencia $|w|_\Sigma$ se llama *mapeo natural*. Entonces: $W^f = \{|w|_\Sigma / w \in W\}$.

- R^f se define como:
  i. si Rwv entonces $R^f|w||v|$, y
  ii. si $R^f|w||v|$ entonces para todo $\Diamond\phi \in \Sigma$, si $\mathcal{M},v \models \phi$ entonces $\mathcal{M},w \models \Diamond\phi$.

La primera de estas condiciones relaciona dos clases de equivalencia cada vez que dos mundos w y v se relacionan en W. La segunda condición se ocupa de conectar dos clases de equivalencia si ocurre que dos mundos se vinculan en W a partir de la semántica pretendida del operador $\Diamond$.

- $V^f(p) = \{|w|_\Sigma / \mathcal{M},w \models p\}$, para todas las letras de proposición p en Σ. Esto es, si una proposición vale en w, en el modelo filtrado la proposición vale en la clase de equivalencia de w, $|w|_\Sigma$. Ello surge naturalmente del concepto de mapeo natural.

Comentario. Por qué el modelo que obtenemos con el filtrado es *finito*: para afirmar ello necesitamos conocer dos resultados. El primero: el conjunto de subfórmulas de una fórmula es claramente finito. En el armado previo, trabajamos con un conjunto Σ que es un conjunto

cerrado (o clausurado) de subfórmulas de ϕ. Dado que trabajamos con subfórmulas de una fórmula bien formada es fácil ver que Σ es finito; Σ se arma así: para todas las fórmulas ϕ, ϕ': i) si $\phi \vee \phi' \in \Sigma$ entonces $\phi \in \Sigma$ y $\phi' \in \Sigma$; ii) si $\neg\phi \in \Sigma$ entonces $\phi \in \Sigma$; y iii) si $\Diamond\phi \in \Sigma$ entonces $\phi \in \Sigma$. El segundo resultado que necesitamos conocer establece que si Σ es un conjunto cerrado de subfórmulas entonces, para algún modelo $\mathcal{M}$, si $\mathcal{M}^f$ es un filtrado de $\mathcal{M}$ a través de un conjunto cerrado Σ de subfórmulas, entonces $\mathcal{M}^f$ contiene a lo sumo 2^n nodos (con n tamaño de Σ). Para probar este resultado recordemos que los mundos de $\mathcal{M}^f$ son las clases de equivalencia $W_\Sigma = \{|w|_\Sigma / w \in W\}$. Sea g una función con dominio W_Σ y rango $\mathcal{P}(\Sigma)$, $g(|w|) = \{ \phi \in \Sigma / \mathcal{M},w \models \phi\}$. A partir de la definición de $\cong_\Sigma$ concluimos que g está bien definida y es inyectiva. Por ello, el tamaño de W_Σ es a lo sumo 2^n, con n tamaño de Σ.

Expresividad. Traducción estándar. Al comienzo de este capítulo mencionamos que la lógica modal puede verse como un fragmento de la lógica de predicados (de primer orden). Trabajamos a continuación esta idea.

El siguiente algoritmo de traducción de fórmulas modales a fórmulas de primer orden nos permite una conexión con un contexto lógico más amplio y bien conocido para nosotros, la lógica de predicados, donde podemos estudiar aspectos de expresividad. El algoritmo ST (por *standard translation* en inglés) recibe una fórmula modal y retorna una fórmula de primer orden con exactamente una variable libre (digamos x). Las fórmulas modales se traducen a fórmulas de primer orden (escritas en un lenguaje de primer orden) que tienen exactamente un símbolo de relación. Intuitivamente, este símbolo de relación se corresponde con la relación que subyace en un frame.

Veamos cómo trabaja el algoritmo. A medida que aparecen operadores modales mientras “parseamos” la fórmula original (esto es, la recorremos sintácticamente de izquierda a derecha), aquéllos se traducirán en variables *nuevas* (que no aparecieron hasta entonces) cuantificadas en la fórmula de salida. A continuación el algoritmo ST:

$ST_x(p) = p(x)$

$ST_x(\bot) = x \neq x$ (una fórmula falsa)

$ST_x(\neg\mathcal{A}) = \neg ST_x(\mathcal{A})$

$ST_x(\mathcal{A} \vee \mathcal{B}) = ST_x(\mathcal{A}) \vee ST_x(\mathcal{B})$

$ST_x(\Diamond\mathcal{A}) = \exists y(Rxy \wedge (ST_y\mathcal{A}))$, donde y es nueva

$ST_x(\Box\mathcal{A}) = \forall y(Rxy \rightarrow (ST_y\mathcal{A}))$, donde y es nueva.

Si, por ejemplo, no estamos en alguna ocasión convencidos del significado intuitivo de una fórmula modal, podemos usar el algoritmo ST y trabajar o analizar la fórmula equivalente en la lógica de predicados.

Ejemplo. Consideremos la fórmula $\Box p \rightarrow \Diamond p$, entonces $ST_x(\Box p \rightarrow \Diamond p) = ST_x(\neg\Box p \vee \Diamond p) = ST_x(\neg\Box p) \vee ST_x(\Diamond p) = \neg ST_x(\Box p) \vee ST_x(\Diamond p) = \neg\forall y(Rxy \rightarrow ST_y(p)) \vee \exists z(Rxz \wedge ST_z(p)) = \neg\forall y(Rxy \rightarrow p(y)) \vee \exists z(Rxz \wedge p(z)) = \forall y(Rxy \rightarrow p(y)) \rightarrow \exists z(Rxz \wedge p(z))$.

El estudio de la expresividad de las fórmulas modales en relación con la lógica de predicados cae en el marco de lo que se conoce como *teoría de correspondencia*. El algoritmo ST es un puente importante entre la lógica modal y la lógica de predicados porque podemos transferir ideas, resultados e incluso algunas técnicas de demostración entre una lógica y otra. Para esto, es útil verificar que no existe distinción matemática entre modelos modales y modelos de primer orden, y que ámbos son esencialmente estructuras relacionales: un modelo modal $\mathcal{M} = (W,R,V)$ provee una relación binaria R que puede usarse para interpretar un símbolo de relación R, y el conjunto $V(p_i)$ puede usarse para interpretar cada predicado unario p_i (correspondiente cada uno de ellos a cada letra de proposición en el lenguaje modal). Dicho esto, existen dos resultados importantes que establecen:

a) *Correspondencia local entre modelos*. Para todo modelo $\mathcal{M}$ y todos los estados $w \in \mathcal{M}$; $\mathcal{M},w \models \mathcal{A}$ si y solo si $\mathcal{M} \models ST_x(\mathcal{A})[w]$ (esta última expresión se lee "la expresión $ST_x(\mathcal{A})$, escrita en un lenguaje de primer orden, es verdadera cuando la variable x se instancia con el valor *w*").
b) *Correspondencia global entre modelos*. Para todo modelo $\mathcal{M}$; $\mathcal{M},w \models \mathcal{A}$ si y solo si $\mathcal{M} \models \forall x\, ST_x(\mathcal{A})$.

La prueba de ambos resultados se hace por inducción sobre la estructura de $\mathcal{A}$.

Ejemplo. Es posible usar el algoritmo ST para obtener la compacidad de la lógica modal como corolario de la prueba de compacidad para la lógica de predicados. La propiedad de compacidad establece que un conjunto Γ de fórmulas de una lógica Λ es Λ-consistente si y solo si todo subconjunto finito de Γ lo es. Para demostrar una de las dos implicaciones, consideremos a Γ un conjunto de fórmulas modales en el que cada subconjunto es satisfactible. ¿El conjunto Γ es satisfactible? Consideremos el conjunto $\{ST_x(\mathcal{A}) \,/\, \mathcal{A} \in \Gamma\}$: es un conjunto de fórmulas escritas en un lenguaje de primer orden. Como cada subconjunto finito de Γ tiene un modelo, por correspondencia local entre modelos sucede que todo subconjunto finito de $\{ST_x(\mathcal{A}) \,/\, \mathcal{A} \in \Gamma\}$ también; y por lo tanto, por *compacidad en primer orden* (probada por ejemplo por A. Hamilton) ese conjunto de fórmulas es satisfactible en algún modelo, digamos $\mathcal{M}$. Entonces, nuevamente por correspondencia local entre modelos, Γ es satisfactible en $\mathcal{M}$.

Lógica deóntica

La lógica deóntica es la "lógica de lo que debe ser", de lo *obligatorio* y lo *prohibido*, y como tal es fundamento de la Ética y del Derecho (*deon* viene del griego *lo que debe ser*). Últimamente se la usa también en el área de la informática para la especificación, por ejemplo, de sistemas y protocolos de seguridad, donde hay permisos y prohibiciones de acceso. La lógica deóntica sienta las bases para el estudio de teorías de argumentación, de lógicas de la acción, de agentes, de grupos; y para el abordaje de enfoques cognitivos del Derecho. Todas estas teorías incluyen novedosas y precisas definiciones formales de conceptos tales como poder institucional, representación, obligaciones, grupos y equipos, delegación, cumplimiento y violación de normas, confianza, contratos, entre otros, con miras a ser aplicadas en sistemas computacionales inteligentes.

Una de las principales características de las reglas deónticas es que pueden ser *violadas*. Es en este aspecto en el que difieren de otras reglas, normas, o principios, por ejemplo de las matemáticas o de la naturaleza. En esos contextos los principios no pueden quebrarse fácilmente. Por ejemplo, a ninguno de nosotros nos tomará demasiado esfuerzo violar la norma que establece que no debemos cruzar el semáforo en rojo cuando conducimos un automóvil; sin embargo, es imposible que un círculo tenga un área distinta a πr^2 o que dos moléculas de hidrógeno y una de oxígeno se unan para formar una sustancia distinta del agua.

Para los informáticos, conocer formalismos simbólicos de la lógica deóntica aumenta nuestras capacidades de razonamiento abstracto en el área de sistemas, y nos prepara para enfrentar desde un punto de vista lógico formal muchas de las modernas teorías de sistemas donde intervienen múltiples agentes, cada uno con sus propias creencias e intenciones, y que interactúan entre ellos para lograr sus objetivos en un ambiente donde hay normas de diferentes tipos y jerarquías. Las aplicaciones de la lógica deóntica a la informática normalmente se relacionan con modos de especificación computacional de normas, esto es, con formas de especificación de comportamiento ideal, de lo que "debe ser". Hay normas que regulan el funcionamiento de los sistemas de computación, el comportamiento, movimiento y seguridad de sus usuarios, y normas que gobiernan el núcleo central de procesamiento de un sistema, propiamente dicho. Los sistemas –y las organizaciones, o instituciones– junto con sus partes, sus usuarios y sus miembros integrantes (que pueden ser otras instituciones), están cruzados por normas de diferentes clases: en los sistemas hay normas de accesos y permisos, hay especificaciones de políticas de trabajo, de comunicación y de acción; hay otros tipos de reglas como las de orden y de limpieza, o guías de comportamiento, horarios de entradas, salidas, hay también restricciones de integridad y de seguridad, hay reglas que son para los usuarios y otras que son para empleados, etc. Cómo modelar computacionalmente normas, cómo hacerlas cumplir, cómo detectar su violación y cómo determinar y exigir un resarcimiento ante un incumplimiento, son temas de los que se ocupa el área de sistemas normativos dentro del área más grande de sistemas inteligentes.

Es conveniente que conozcamos las dificultades que tiene la lógica proposicional básica para capturar formalmente el discurso normativo, donde cobra especial interés la categoría filosófica del "deber ser". La lógica proposicional clásica es insuficiente para representar dicha categoría porque las proposiciones son o verdaderas o falsas, es decir, las cosas son o no son. No podemos simbolizar que las cosas "deben ser" o "está prohibido que sean", solo podemos simbolizar la *forzosidad* de que las cosas son o no son, que los hechos ocurren o no ocurren. Mostraremos esta incapacidad de la lógica proposicional para representar el "deber ser" con un caso simple (ejemplo de la Biblioteca del Imperial College, dado por A. Jones y citado por R. Wieringa y J. Meyer):

Ejemplo. Reglas de la biblioteca. *El lector devolverá el libro en 15 días hábiles. Si el lector devuelve el libro en 15 días hábiles, no se le aplicará el apercibimiento administrativo del artículo 20. Si el lector no devuelve el libro en 15 días hábiles, se le aplicará el apercibimiento administrativo del artículo 20.*

Si formalizamos estas reglas usando lógica proposicional, tenemos tres proposiciones: p, $p \rightarrow \neg q$ y $\neg p \rightarrow q$ para la primera, segunda y tercera regla de la biblioteca, respectivamente. Supongamos que ocurre que el lector no devuelve el libro en 15 días hábiles; formalizamos ese hecho como $\neg p$. Tenemos entonces: i) una contradicción entre este hecho nuevo y la primera regla de la biblioteca; y también tenemos ii) entre las dos primeras reglas deducimos $\neg q$ por *modus ponens*, y entre la tercera regla y el hecho nuevo $\neg p$ deducimos q, con lo cual conseguimos q y $\neg q$. Esto sorprende, porque las reglas tal como están presentadas en lenguaje natural son coherentes desde el punto de vista de lo que se debe hacer para el correcto funcionamiento de la biblioteca, y porque además las hemos traducido de un modo directo al lenguaje de la lógica proposicional. No solo no hay error alguno en las reglas de la biblioteca ni en su formalización proposicional, sino que tampoco hay error en el proceso de deducción llevado a cabo. Simplemente, la lógica proposicional "se queda corta" para representar que algo es forzoso.

Esta dificultad de la lógica proposicional para modelar el deber ser favoreció la búsqueda de representaciones que fueran adecuadas.

Formalización de conceptos deónticos. El aspecto técnico más influyente sobre las descripciones formales modernas de la lógica deóntica aparece en el trabajo seminal de G. von Wright, *Deontic Logic*, de 1951. Dicho trabajo define un sistema formal proposicional elemental que incluye los modos deónticos básicos P, F y O de *permiso*, *prohibición* y *obligación*, representados con tablas de verdad que incluyen los conectivos booleanos usuales. Las letras proposicionales se corresponden con acciones simples. Por ejemplo, con la letra p describimos acciones tales como "pagar", "estacionar", "matar", "adeudar", "fumar", "robar", etc. Asumimos que la expresión negada $\neg p$ se interpreta como "no estar haciendo p". Conceptualmente entonces tenemos que los modos deónticos *modalizan acciones*. Algunos renglones de las

tablas, por corresponderse con escenarios imposibles para un contexto normativo, no eran para von Wright combinaciones admisibles. Por ejemplo, al armar la tabla de verdad de la proposición $Pp \vee P\neg p$, cuya lectura intuitiva es "está permitido hacer p o está permitido no hacer p", la combinación falso-falso de valores de verdad para Pp y para $P\neg p$ es una combinación inadmisible porque o bien uno tiene permitido hacer p o bien uno tiene permitido hacer $\neg p$; en la realidad no se da el caso de que *poder hacer p* y *poder hacer la negación de p* sean ambas falsas, pues en un momento dado o estamos haciendo p o no lo estamos haciendo. Entonces, es por ello que el renglón falso-falso en la tabla de verdad de la proposición $Pp \vee P\neg p$, para von Wright no existe.

Luego de que von Wright explicara su sistema deóntico en 1951, el área de la lógica deóntica floreció de estudios y se descubrió que aquellos tres operadores para los cuales von Wright armaba tablas de verdad podían –con mayor o menor éxito– representarse con los operadores de necesidad y posibilidad de una lógica modal normal.

En el resto de esta sección estudiamos a la lógica deóntica descripta como una lógica modal normal.

Operadores modales en su interpretación deóntica. El operador "$\Box$" es comúnmente usado para modalidades de carácter universal; esta intuición ya la manejamos porque conocemos la semántica del operador $\Box$. Notemos que esta universalidad del $\Box$ coincide con nuestra idea básica de obligatoriedad: algo es obligatorio si necesariamente debe ser cumplido sea cual sea la situación o el estado de cosas.

Entonces, para trabajar en un contexto deóntico, simplemente reescribimos $\Box$ como O (por "obligatorio"). Así, tenemos que dado un mundo *w* la fórmula $O\mathcal{A}$ es verdadera en *w* si $\mathcal{A}$ es verdadera en todos los mundos o situaciones que son adyacentes a *w*: la semántica para O es *verdad en todos los mundos R-accesibles*.

El dual del operador O es el operador P cuya lectura intuitiva es "permitido": Notemos que la dualidad $Op \leftrightarrow \neg P\neg p$ se ajusta a nuestra intuición de permiso pues algo es obligatorio si no ocurre que está permitido que su negación suceda. Por ejemplo, "es obligatorio hacer silencio" es equivalente a "no se permite no hacer silencio".

Finalmente, el tercer operador comúnmente usado en la lógica deóntica es una abreviatura: definimos el operador F, cuya lectura intuitiva es "prohibido", como $Fp \leftrightarrow \neg Pp$, esto es, si algo está prohibido es porque no está permitido que lo llevemos a cabo. Por ejemplo "prohibido fumar" es equivalente a "no se permite fumar". Es fácil ver que Fp también puede escribirse como $O\neg p$: reemplazando adecuadamente vemos que $O\neg p \equiv \neg P\neg(\neg p) \equiv \neg Pp \equiv Fp$.

A continuación describimos formalmente desde el punto de vista sintáctico un sistema modal básico de lógica deóntica.

Definición 3.13. Sistema modal KD de la lógica deóntica. Definimos el sistema de la lógica deóntica como sigue:

Lenguaje

El lenguaje proposicional estándar, al que sumamos los símbolos O, P, F.

Axiomas

Todas las tautologías de la lógica proposicional.

(K) $O(p \rightarrow q) \rightarrow (Op \rightarrow Oq)$

(P) $Pp \leftrightarrow \neg O \neg p$

(D) $Op \rightarrow Pp$

(F) $Fp \leftrightarrow O\neg p$

Reglas de inferencia

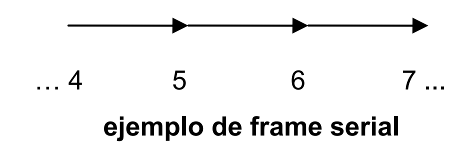
modus ponens, generalización y sustitución uniforme.

Algunos comentarios sobre esta definición. La lógica deóntica así definida –como una extensión de la lógica modal normal mínima– captura la intuición que tenemos sobre el universo de las normas. El axioma K es el axioma de distribución de toda lógica normal. Los axiomas (P) y (F) se corresponden con el dual de O y con una abreviatura, respectivamente. El axioma (D) establece el principio deóntico de que si algo es obligatorio entonces está permitido, lo que tiene sentido, pues si pretendemos que algo sea impuesto por una norma entonces ese algo tiene que estar permitido o habilitado.

Discusión. Notemos que podemos aplicar la regla de generalización a las tautologías proposicionales por ser éstas teoremas del sistema de la lógica deóntica. Pero, ¿las tautologías son obligatorias?

Discusión conexa. La fórmula $\neg O\bot$ es un teorema del sistema formal de la lógica deóntica. Para probarlo, veamos que la fórmula $O\top$ es demostrable en el sistema por aplicación de la regla de generalización (con $\top$ constante *true*, definida como abreviatura, tal como hicimos en la sección anterior). Seguidamente, invocamos el axioma (D) y conseguimos $O\top \rightarrow P\top$. Aplicando *modus ponens* y a continuación (P) obtenemos $P\top \equiv \neg O \neg \top \equiv \neg O\bot$. Este teorema le da cierta coherencia fundamental a cualquier sistema de normas, impidiéndole tener normas que sean contradicciones.

Semántica. A la semántica formal de esta lógica deóntica se la llama KD (por abuso de lenguaje se suele mencionar el nombre del cálculo sintáctico, normalmente en la jerga se dice: "este sistema tiene semántica KD estándar"). Los modelos para los frames tienen estructura (W, R_O, V), con W los mundos, V la función de valuación y R_O la relación de accesibilidad tal que cumple que para todo $w \in W$ existe un $v \in W$ tal que $R_O wv$ ($\forall w \exists v : R_O wv$). Esto es, la lógica deóntica presentada es fuertemente completa respecto de los frames *seriales* o "sin límite a derecha"; son frames en los que la relación de accesibilidad entre mundos se denomina *serial*: siempre para cada mundo hay otro mundo accesible.



ejemplo de frame serial

Para probar que la lógica KD es fuertemente completa respecto de los frames sin límite a derecha es suficiente mostrar que el modelo canónico para KD es sin límite a derecha. Esto requiere de una prueba de existencia. Sea *w* cualquier mundo en el modelo canónico para KD, debemos probar que existe un *v* tal que $R_{KD}wv$. Como *w* es un conjunto KD-maximal consistente, entonces contiene la fórmula $\Box p \rightarrow \Diamond p$. Por lo tanto, por clausura de los conjuntos maximales consistentes y por sustitución uniforme, *w* contiene a $\Box \top \rightarrow \Diamond \top$ (con $\top$ constante *true*). Como las tautologías pertenecen a toda lógica modal normal, por aplicación de la regla de generalización $\Box \top$ también, y entonces, por *modus ponens*, $\Diamond \top \in w$, y por lo tanto existe *v* sucesor R_{KD}-accesible de *w* por aplicación del *lema de existencia*. Este lema establece que para toda lógica normal Λ y cualquier estado $w \in W^{\Lambda}$, si $\Diamond \mathcal{A} \in w$ entonces existe un estado $v \in W^{\Lambda}$ tal que $R^{\Lambda}wv$ y $\mathcal{A} \in v$, con W^{Λ} y R^{Λ} como en la definición 3.12.

Algunos teoremas de KD. Algunos de los más relevantes son:

(O∧) $O(p \wedge q) \leftrightarrow (Op \wedge Oq)$

(P∧) $P(p \wedge q) \rightarrow (Pp \wedge Pq)$

(F∧) $(Fp \vee Fq) \rightarrow F(p \wedge q)$

(O∨) $(Op \vee Oq) \rightarrow O(p \vee q)$

(P∨) $P(p \vee q) \leftrightarrow (Pp \vee Pq)$

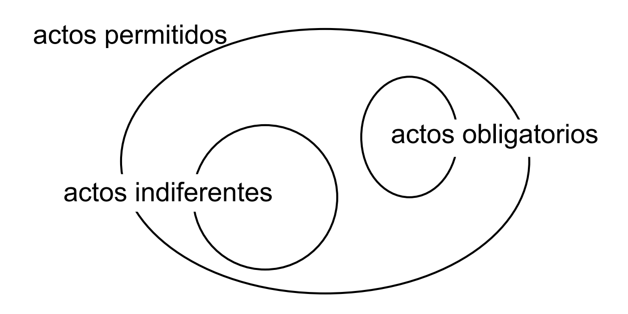
(F∨) $F(p \vee q) \leftrightarrow (Fp \wedge Fq)$

Es importante notar que en estos teoremas hay implícita una suposición de *cotemporalidad*, es decir, los actos o hechos representados por las letras proposicionales en cada teorema se consideran como ocurriendo *a la vez*, *en simultáneo*. Vemos que la obligación de una conjunción de, digamos, dos actos, es equivalente a la conjunción de las obligaciones de cada acto por separado (O∧), y que el permiso de una disyunción es equivalente a la disyunción de los permisos (P∨). Ambas equivalencias seguramente nos resultan intuitivas. Para (O∧) por ejemplo, una posible lectura en lenguaje natural es: “es obligatorio permanecer de pie” y “es obligatorio guardar silencio”, y “es obligatorio permanecer de pie y guardar silencio”. Siguiendo, notemos que el permiso de una conjunción implica la conjunción de los permisos (P∧); pero en el otro sentido la implicación no vale: por ejemplo, que esté permitido conducir un automóvil y también esté permitido hablar por teléfono móvil no implica que estén permitidas a la vez ambas acciones. Dejamos al lector la lectura intuitiva (y también la demostración) de los teoremas restantes teniendo en cuenta la suposición de cotemporalidad.

Ventajas del enfoque modal de la lógica deóntica. La descripción lógica de normas usando operadores deónticos permite descubrir patrones normativos. A partir de allí, es posible explorar nuestras capacidades de diseño y análisis de normas a distintos niveles de jerarquías de normas y también en cuanto a normas de distintos tipos: jurídicas, morales, de tránsito, de etiqueta, etc. Adquirimos así mayor capacidad "ingenieril" en el sentido de poder definir y determinar las formas lógicas de normas, relaciones entre ellas, y nuevas categorías normativas.

Ejemplo. Definición de nuevas categorías normativas. von Wright dio la definición para el concepto de *acto indiferente* usando el operador deóntico P del siguiente modo: un acto (simbolizado con una letra proposicional) es indiferente si el acto está permitido y su negación también. Por ejemplo, en una plaza está permitido fumar, y también está permitido no fumar (en símbolos: $Pf \wedge P\neg f$). Además, von Wright explicó que, si bien todos los actos indiferentes están permitidos ($(Pf \wedge P\neg f) \rightarrow Pf$), aquello que está permitido no es indiferente (por ejemplo, que esté permitido honrar a la patria no implica que esté permitido honrar a la patria y que también esté permitido deshonrarla). Los actos indiferentes pueden lucir triviales en su estructura lógica; sin embargo, pueden resultar relevantes para la tarea de diseño de sistemas normativos, esto es, en un sentido "ingenieril" de un cuerpo normativo: cuando ciertos actos son identificados como indiferentes, seguramente no integrarán ninguna obligación, no serán parte de ninguna norma. Al identificar actos indiferentes podemos "purgar" o "limpiar" un cuerpo normativo (una base de datos normativa) de ellos. von Wright también sostuvo que lo que es obligatorio está permitido (axioma D) pero no es indiferente; ello es fácil de ver porque por el axioma D tenemos $Op \rightarrow Pp$, pero no es posible derivar $Op \rightarrow (Pp \wedge P\neg p)$ en el sistema.

Notemos que estas estructuras de fórmulas, que representan diferentes categorías normativas, son aplicables a actos considerados aisladamente, "de a uno". von Wright presentó también conceptos deónticos que se aplican a pares de actos, como la idea de *actos incompatibles*: dos actos son incompatibles si su conjunción está prohibida: $F(p \wedge q)$, como por ejemplo conducir un automóvil y hablar por teléfono móvil. También presentó la idea de *compromiso*: un acto nos compromete a (hacer) otro acto si la implicación entre ámbos es obligatoria. Por ejemplo, hacer una promesa nos compromete a cumplirla: vemos que $O(p \rightarrow q) \equiv \neg P\neg(p\rightarrow q) \equiv \neg P\neg(\neg p \vee q) \equiv \neg P(p \wedge \neg q)$, que puede leerse intuitivamente como "si uno se obliga prometiendo que si p es el caso, entonces cumplirá con q". Vemos que no está permitido prometer p y no cumplir con q ($\neg q$). En resumen, gráficamente tenemos:



Paradojas deónticas. Las paradojas deónticas son expresiones que son verdaderas en el sistema KD pero que carecen de significado o son, directamente, contradictorias cuando las analizamos desde el sentido común. Algunos ejemplos son:

Paradoja de Ross	$Op \rightarrow O(p \vee q)$
Paradoja del penitente	$Fp \rightarrow F(p \wedge q)$
Obligación derivada	$Op \rightarrow O(q \rightarrow p)$
Sistema normativo vacío	$O\top$, con $\top$ *true* (cualquier tautología)

Un ejemplo en lenguaje natural de la paradoja de Ross es "es obligatorio que lleves esta carta al correo, entonces es obligatorio que o lleves esta carta al correo o la quemes". La paradoja del penitente puede ejemplificarse en lenguaje natural con "está prohibido matar, por lo tanto están ámbos prohibidos matar y arrepentirse". Notar que la paradoja de la obligación derivada proviene de la definición de la función de verdad del condicional tal como lo conocimos al estudiar la lógica proposicional (que en la jerga algunos denominan "implicación material").

von Wright consideró que las tautologías no necesariamente debían ser obligatorias, y que tampoco las contradicciones deben estar prohibidas. Estos dos escenarios deónticos le resultaban a von Wright innecesarios, ajenos a cualquier sistema normativo que se preciara de ser coherente con la realidad, fundando esto en el hecho de que no estamos obligados a hacer cosas verdaderas y en que muchas veces hacemos contradicciones. A estos dos escenarios los consideró integrantes de lo que llamó el *principio de contingencia deóntica*, que podemos formalizar con: $\neg O\top \wedge \neg F\bot$. Ahora bien, notemos que tal como lo hemos planteado en la discusión conexa a la definición 3.13, la fórmula $O\top$ es teorema del sistema formal KD, y también notemos que $O\top \equiv (\neg P \neg)\top \equiv F(\neg\top) \equiv F\bot$. Justamente, las fórmulas $O\top$ y $F\bot$ hacen caer el principio de contingencia deóntica considerado válido por von Wright, para quien $O\top$ y $F\bot$ no deben ser teoremas de ningún sistema de normas. Así, vemos que KD entra en colisión con el sistema original propuesto por von Wright.

Decidibilidad del sistema KD. Dejamos al lector la prueba de decidibilidad de KD, teniendo en cuenta que ya sabemos que: KD es axiomatizable mediante un número finito de esquemas de axioma, y que el axioma D determina la clase de los frames seriales (lo hemos probado más arriba). Queda por demostrar, para la prueba de decidibilidad, que KD posee la propiedad de modelo finito. Ello puede hacerse mediante un filtrado, siguiendo los pasos descriptos en la sección anterior.

Otros enfoques para la representación de normas. El enfoque que usa la lógica proposicional clásica –despojada de operadores deónticos, como en el ejemplo de las reglas de la biblioteca– se ubica en lo que en el área se denomina "enfoque factual", es decir, relativo

a los hechos, a “lo que es”. A este enfoque también pertenecen intentos de representar normas usando lógica de predicados –tal como la hemos estudiado en el segundo capítulo–. En este contexto, las normas se ven como *definiciones* en lugar de obligaciones, permisos y prohibiciones. De este modo, las normas se formalizan como predicados de un lenguaje de primer orden, o como cláusulas en un programa Prolog. Esta versión del diseño de normas presenta las conocidas ventajas y características que posee la lógica de predicados por sobre la lógica proposicional; pero tengamos en cuenta que las cláusulas Prolog o los predicados de primer orden no permiten capturar la categoría deóntica, es decir, no permiten diferenciar entre lo que “es” y lo que “debe ser”, sino que *formalizan conceptos*. Autores como Jones han remarcado que, claramente, los enfoques factuales de las normas son limitados en su capacidad de modelización, pero *no tienen nada de malo* dado que permiten estudiar, por ejemplo, cómo diferentes definiciones o conceptos legales se aplican a un caso en estudio, cómo analizar textos legales, etc.

Operadores deónticos relativizados. Hasta aquí hemos visto a la lógica deóntica que tiene un operador modal O de obligación. Este operador O es *genérico* en el sentido de que es *impersonal*, pues asume una *referencia tácita a todos los obligados*, que somos *todos los individuos* o agentes integrantes del grupo o de la comunidad de que se trate. Así, entendemos al operador O como de *obligación general*. Pero podemos hacer una distinción y referirnos a quiénes son los individuos obligados, introduciendo operadores de obligación relativizados a dichos individuos, operadores que autores como H. Herrestad y C. Krogh llaman *obligaciones especiales*. Estos autores definen un nuevo operador deóntico $O_x\mathcal{A}$ cuya lectura intuitiva es “es obligatorio $\mathcal{A}$ para el individuo o agente *x*” (y su semántica es KD). Supongamos que consideramos la posibilidad de crear una extensión del sistema KD agregándole esta modalidad. Entonces aparecen relaciones interesantes que debemos considerar. Por ejemplo, seguramente pretenderemos que valga en la extensión el axioma $O\mathcal{A} \rightarrow O_x\mathcal{A}$ que establece que si algo es obligatorio en términos generales entonces es obligatorio para el individuo *x*. Del mismo modo que relativizamos obligaciones para denotar normas individuales, podemos relativizar el operador deóntico para que indique a favor de qué individuo debe *x* cumplir la obligación de $\mathcal{A}$. Por ejemplo, podemos definir $O^y{}_x\mathcal{A}$ cuya lectura intuitiva es “*x* está obligado a $\mathcal{A}$ en el interés de *y*”. Este tipo de obligación relativizada es muy específica pues denota las dos partes: el deudor y el beneficiario de una obligación individual.

Estudiamos en la sección siguiente más aspectos referidos a operadores individuales para cada agente.

Síntesis. Hemos mencionado a la lógica proposicional y su limitación para representar satisfactoriamente la categoría del “deber ser” de las normas. Hemos estudiado cómo el enfoque modal deóntico sí hace una distinción precisa entre las categorías filosóficas del “deber ser” (lo ideal) y del “ser”. Muchas veces las distinciones y debates filosóficos no tienen aplicaciones concretas en la realidad; la formalización de la lógica deóntica como una lógica

modal computable es un ejemplo de un concepto filosófico que puede materializarse y ser puesto en uso en un sistema computacional.

Sistemas multiagente

El área de sistemas multiagente (MAS, por las siglas en inglés de *multi-agent systems*) se ocupa principalmente de modelar agentes cognitivos (actores humanos o entidades computacionales que *saben* y *conocen*) o reactivos (que *actúan* y *reaccionan*), que dependen unos de otros para lograr sus objetivos individuales o grupales, e interactúan en varios y diferentes ambientes.

Hay al menos cuatro usos actuales de sistemas multiagente que describen la segmentación del campo de estudio: i) el diseño de sistemas distribuidos o híbridos, ii) la formulación, simulación y resolución de problemas haciendo foco en unidades sociales, grupos y organizaciones, iii) el desarrollo de teorías socio-filosóficas, y iv) la comprensión de temas sociales y hechos sociales.

MAS hace énfasis en el comportamiento visible de los agentes, en el conocimiento que manejan, en los diferentes tipos de normas que regulan el accionar de los agentes, y en las agrupaciones de agentes que se comportan como unidades sociales de distintas envergaduras. Los agentes artificiales imitan (o intentan imitar) atributos humanos y capacidades humanas que, en el área, se describen con términos provenientes de las ciencias cognitivas: “pensar”, “adaptarse”, “aprender”, “argumentar”; ser “racional”, ser “emotivo”, o “rutinario”. Las estructuras de grupos de agentes y las relaciones entre agentes se describen usando terminología sociológica: “organización”, “comunidad”, “coalición”, “grupo”, “poder”, “solidaridad”, “normas”, “contratos”, “institución”, etc.

A partir de aquí usamos la sigla MAS no solo para referirnos al área de estudio sino también como abreviatura de la expresión “sistema(s) multiagente”, cuando no hay confusiones.

Descripción formal de sistemas multiagente (MAS). La lógica modal es –por su flexibilidad y naturalidad en la escritura– una herramienta ampliamente aceptada para el diseño y el desarrollo de MAS. Con el fin de dar una definición de los estados mentales y cognitivos de los agentes, se formaliza con distintas lógicas modales especiales la postura del agente hacia su entorno: lo que el agente sabe, cuáles son sus creencias, cuáles son sus objetivos, cómo actúa, etc.

Los sistemas más conocidos e influyentes de este tipo son los llamados de *creencia-deseo-intención* BDI (por el inglés *belief-desire-intention systems*). Los agentes BDI se describen a través de: i) un estado “mental” dado en términos de creencias (*beliefs*) correspondientes a la información que el agente tiene sobre el entorno (que “cree” que sucede alrededor); ii) los deseos (*desires*), que son opciones que tiene el agente, y iii) las intenciones (*intentions*) que representan deseos elegidos por el agente (para ser cumplidos, o intentar ser cumplidos). Las

creencias son vistas como “información” del agente. Los deseos e intenciones son vistos como actitudes *motivacionales*, como una inspiración para la actividad del agente.

Para representar cada uno de estos aspectos existen lógicas específicas de poder expresivo limitado, como por ejemplo: una lógica de creencias, una lógica de intenciones, una lógica de objetivos, una lógica del actuar, etc. De manera análoga a la discutida en la presentación de la lógica deóntica, una descripción formal de estas lógicas específicas puede hacerse a través de una lógica proposicional extendida con una colección de operadores modales $\Box_x$ indexada por una colección A de agentes ($\Box_x$ / $x \in A$). Típicamente, para cada $x \in A$, $\Box_x$ funciona como un operador modal normal que satisface axiomas extra que capturan algún aspecto relevante del agente.

Ejemplo. Lógica de creencias. En la lógica de creencias escribimos $Bel_x \mathcal{A}$ por $\Box_x\mathcal{A}$. $Bel_x \mathcal{A}$ es una modalidad epistémica (o “del conocimiento”). Es usada para representar “el agente *x* cree que $\mathcal{A}$”, con $\mathcal{A}$ proposición.

El conjunto de creencias de un agente representa su “estado mental”. Para la lógica de creencias, requerimos: $Bel_x \mathcal{A} \wedge Bel_x(\mathcal{A} \rightarrow \mathcal{B}) \rightarrow Bel_x\mathcal{B}$ (distribución de creencias), $\neg Bel_x\bot$ (consistencia de creencias), $Bel_x\mathcal{A} \rightarrow Bel_x(Bel_x \mathcal{A})$ (introspección positiva), $\neg Bel_x\mathcal{A} \rightarrow Bel_x(\neg Bel_x\mathcal{A})$ (introspección negativa), y *de* $\mathcal{A}$ *se obtiene* $Bel_x\mathcal{A}$ (regla de generalización para creencias). El axioma de consistencia de creencias nos asegura que el agente no cree en contradicciones, esto es, no cree en algo y en lo opuesto. El axioma de introspección positiva afirma que si el agente cree algo entonces cree en lo que cree, y el axioma de introspección negativa establece que si un agente no cree en algo entonces cree que no cree ese algo. Finalmente, la regla de generalización establece que el agente cree en algo si ese algo pudo probarse como cierto.

Ejemplo. Lógica de objetivos. En la lógica de objetivos escribimos $Goal_x \mathcal{A}$ por $\Box_x\mathcal{A}$. La expresión $Goal_x \mathcal{A}$ representa “el agente *x* tiene el objetivo $\mathcal{A}$”, con $\mathcal{A}$ proposición, que refleja algún estado particular de cosas (por ejemplo: “viajar”) que el agente quiere llevar a cabo. Para la lógica de objetivos requerimos: $Goal_x\mathcal{A} \wedge Goal_x(\mathcal{A} \rightarrow \mathcal{B}) \rightarrow Goal_x\mathcal{B}$ (distribución de objetivos) y *de* $\mathcal{A}$ *se obtiene* $Goal_x\mathcal{A}$ (regla de generalización para objetivos).

Ejemplo. Lógica de intenciones. En la lógica de intenciones escribimos $Int_x \mathcal{A}$ por $\Box_x\mathcal{A}$. La expresión $Int_x \mathcal{A}$ significa “el agente *x* tiene la intención de que $\mathcal{A}$ sea verdadero”, con $\mathcal{A}$ proposición. Las intenciones son vistas en MAS como inspiración para actividades. Para la lógica de intenciones requerimos: $Int_x \mathcal{A} \wedge Int_x(\mathcal{A} \rightarrow \mathcal{B}) \rightarrow Int_x \mathcal{B}$ (distribución de intenciones), $\neg Int_x\bot$ (consistencia de intenciones) y *de* $\mathcal{A}$ *se obtiene* $Int_x \mathcal{A}$ (regla de generalización para intenciones). Las intenciones son objetivos seleccionados por el agente para intentar convertirse en verdaderos.

Comentarios. Bel_x, $Goal_x$ e Int_x capturan la configuración *interna* de un agente. Notemos que estas tres lógicas tienen en su descripción, cada una, una instancia del esquema general de distribución $\Box\mathcal{A} \wedge \Box(\mathcal{A} \rightarrow \mathcal{B}) \rightarrow \Box\mathcal{B}$ que define cierto grado básico de "racionalidad" (notemos que el axioma de distribución guarda una estructura parecida a la de la regla *modus ponens*). La lógica de creencias y la de intenciones tienen ambas el axioma de consistencia, pero la de objetivos no. Esto se asemeja bastante a lo que nos sucede usualmente a los seres humanos, que podemos (y solemos) tener objetivos contradictorios; pero cuando elegimos objetivos para que sean nuestras intenciones e intentar concretarlas hacemos esa elección de intenciones de modo tal que no se contradigan entre sí.

Notemos que, tal como está presentada, la lógica de objetivos no es más que una colección de modalidades con semántica K básica.

En el área de MAS normalmente se asimila la noción de "lo que el agente cree" con aquélla de "lo que el agente sabe", como un modo de establecer que el agente efectivamente cree en lo que sabe, esto es, que lo que cree y lo que sabe son lo mismo. Sin embargo, en otras ocasiones, cuando hay que distinguir con precisión entre lo que el agente sabe y lo que el agente cree, se usa la lógica llamada epistémica (vimos un ejemplo al principio de este capítulo) que usa la modalidad $K_x\mathcal{A}$, para representar "el agente sabe $\mathcal{A}$" (la K proviene del inglés *knows*).

A continuación presentamos el operador $Does_x$. A diferencia de las tres modalidades anteriores, este operador indica el actuar visible, *externo*, de un agente:

Ejemplo. Lógica de la acción. $Does_x\mathcal{A}$ representa actividad exitosa del agente *x*. Su lectura intuitiva es: "el agente *x* lleva a cabo la acción $\mathcal{A}$", con $\mathcal{A}$ proposición. La lógica del Does en su definición axiomática tiene los esquemas: $Does_x\mathcal{A} \rightarrow \mathcal{A}$, $(Does_x\mathcal{A} \wedge Does_x\mathcal{B}) \rightarrow Does_x (\mathcal{A} \wedge \mathcal{B})$, y $\neg Does_x\mathrm{T}$ (con T abreviatura de *true*). El primer axioma establece efectividad en el actuar: si el agente x lleva a cabo la acción $\mathcal{A}$, entonces $\mathcal{A}$ sucede. El segundo axioma, conocido como *axioma de aglomeración*, se refiere a la cotemporalidad implícita en la lógica modal proposicional: si el agente lleva a cabo la acción $\mathcal{A}$ y lleva a cabo la acción $\mathcal{B}$ entonces el agente lleva a cabo las dos acciones en el mismo tiempo (por ejemplo, "hizo el backup del dispositivo y lo apagó"). El tercer axioma de algún modo formaliza la idea de que un agente lleva a cabo acciones que son plausibles y, principalmente, evitables. La noción de acción es de algún modo un concepto de control: ningún agente lleva a cabo acciones inevitables, y las tautologías son inevitables (intuitivamente podemos asumir que las tautologías "se hacen solas" sin la intervención de ningún agente).

Observación. Si bien D. Elgesem acepta el axioma de aglomeración, no acepta el esquema inverso, llamado M: $Does_x(\mathcal{A} \wedge \mathcal{B}) \rightarrow (Does_x \mathcal{A} \wedge Does_x \mathcal{B})$. Esto porque, en presencia de sustitución uniforme por equivalentes lógicos, a partir de $Does_x \mathcal{A}$ podemos conseguir la equivalencia $(Does_x \mathcal{A}) \leftrightarrow (Does_x \mathcal{A} \wedge (\mathcal{B} \vee \neg\mathcal{B}))$, y si aplicamos el axioma M obtenemos

$\text{Does}_x (\mathcal{A} \wedge (\mathcal{B} \vee \neg\mathcal{B})) \rightarrow \text{Does}_x \mathcal{A} \wedge \text{Does}_x (\mathcal{B} \vee \neg\mathcal{B})$ y entonces conseguimos $\text{Does}_x \mathrm{T}$ que es para Elgesem contraintuitivo en una lógica de la acción, contradice el axioma $\neg\text{Does}_x \mathrm{T}$.

Además, M junto a la generalización y sustitución de equivalentes nos da una instancia de la paradoja de Ross, tal como la vimos al estudiar la lógica deóntica: $\text{Does}_x \mathcal{A} \rightarrow \text{Does}_x (\mathcal{A} \vee \mathcal{B})$, que es inaceptable en la intuición de una lógica de la acción. Dejamos al lector esta comprobación.

Lógicas no normales. En su definición de la lógica de la acción, Elgesem explica que Does no puede ser un operador normal porque si lo fuera entonces su comportamiento no sería el que esperamos para representar acciones. Veamos: si la lógica del Does fuese normal entonces fácilmente la podríamos definir como una extensión de K agregándole a K el axioma de éxito $\text{Does}_x \mathcal{A} \rightarrow \mathcal{A}$ (es decir, agregándole a K el esquema modal llamado T) pues es en virtud de este único axioma que la lógica refleja lo que esperamos del actuar de un agente: éxito y control en el actuar. Entonces adoptaríamos el sistema normal KT para la lógica de la acción. Ahora bien, por ser normal entonces la lógica del Does verificaría la regla de generalización *si* $\vdash \mathcal{A}$ *entonces* $\vdash \textit{Does}_x \mathcal{A}$ que nos lleva a derivar, dentro del sistema, la fórmula $\text{Does}_x\mathrm{T}$ (con T abreviatura de *true*) que, hemos dicho, no queremos que sea verdadera pues las cosas que son factibles de ser hechas tienen que ser evitables y no tautologías. Con lo cual la regla de generalización no es deseable para definir una lógica de la acción. Además, si la lógica del Does fuese normal, verificaría también el axioma K de distribución: $\text{Does}_x (\mathcal{A} \rightarrow \mathcal{B}) \rightarrow (\text{Does}_x\mathcal{A} \rightarrow \text{Does}_x\mathcal{B})$ lo que nos permitiría derivar, por ejemplo, proposiciones indeseadas para una lógica de la acción, tal como $\text{Does}_x\mathcal{A} \rightarrow \text{Does}_x (\mathcal{A} \vee \mathcal{B})$ (esto a partir del teorema $\mathcal{A} \rightarrow (\mathcal{A} \vee \mathcal{B})$, generalización, axioma K de distribución y *modus ponens*). Por todo esto es que Elgesem decide que la lógica del Does no puede tener una semántica modal normal.

Definición 3.14. Lógica modal no normal. Una lógica modal es *no normal* cuando no satisface el axioma K de distribución (ver definiciones 3.7.b y 3.8).

Así las cosas, los modelos de Kripke no son suficientes para dar una semántica de lógicas no normales. Tenemos entonces una semántica diferente, llamada *de tipo Scott-Montague*. La intuición detrás de esta semántica es la siguiente: en lugar de tener una relación entre mundos, tenemos un conjunto de colecciones de mundos conectados a *w*. Esas colecciones se llaman *neighbourhoods* o vecindarios de *w*.

Formalmente: un frame de Scott-Montague es un par ordenado (W,N), donde W es un conjunto (de mundos, puntos, situaciones, etc.) y N es una función que asigna a cada elemento $w \in$ W un conjunto de subconjuntos de W (los neighbourhoods de *w*). Un modelo de Scott-Montague es una terna (W,N,V), donde (W,N) es un frame de Scott-Montague y V es una función de valuación como en los modelos de Kripke, y la definición de verdad en *w* es: $\Box\mathcal{A}$ es verdadera en *w* si y solo si los elementos de W donde $\mathcal{A}$ es verdadera son los conjuntos de N(*w*).

Esta es una generalización de la semántica tradicional de Kripke. Es fácil ver que un frame de Kripke es equivalente a un frame de Scott-Montague con el mismo W y donde N(*w*) se define como $\{v \,/\, wRv\}$. Debe quedar claro que, por el contrario, hay frames de Scott-Montague que no se corresponden con frames de Kripke. Además, lamentablemente, hemos perdido el paralelismo entre □ y ◊ y los cuantificadores universal y existencial respectivamente, paralelismo que sí existe con esos cuantificadores en los frames de Kripke.

Lógicas de propósitos especiales. Comentarios. En los sistemas BDI la actividad de un agente comienza a partir de objetivos. Un agente tiene en general muchos objetivos, muchos de los cuales no serán perseguidos, no estarán relacionados con acciones. Esto permite que un agente pueda comportarse consistentemente aún cuando tenga objetivos inconsistentes. Un agente elige un número finito de sus objetivos para que sean sus intenciones, es decir, sus motivaciones para actuar. No nos resulta relevante cómo una intención se forma a partir de un conjunto de objetivos, solo nos concentramos en el hecho de que los objetivos son elegidos por el agente de modo tal que se preserve consistencia (ver el axioma de consistencia de intenciones). Un problema con las lógicas modales estándar para creencias (y conocimiento) es que los agentes son formalizados como *omniscientes*: creen en todos los teoremas así como en las consecuencias lógicas de sus creencias. Cualquier lógica modal estándar con semántica de Kripke en la que se modela creencia como un operador de necesidad tendrá esta propiedad. El problema aquí es que la omnisciencia no se aplica a los humanos, que tenemos normalmente poco tiempo disponible y racionalidad limitada, es irreal asumir que creemos en cada teorema (¡los hay muy complicados!). Finalmente, la lógica del Does no satisface el inverso del axioma de aglomeración. Si lo hiciese, en presencia del axioma de aglomeración y de la regla de sustitución uniforme la lógica de la acción se volvería inconsistente. Dejamos al lector la comprobación de ello.

Discusión. Dejamos al lector la lectura intuitiva de cada uno de los axiomas de las lógicas descriptas para creencias, objetivos, intenciones y acción.

Creación de sistemas multimodales multiagente BDI. El mecanismo es, desde el punto de vista ingenieril, el siguiente: se seleccionan diferentes lógicas modales específicas, también llamadas de *propósitos especiales* o *de propósito determinado*, que son –casi siempre– monomodales, es decir, con un único operador modal, con o sin su dual. Se las *combina* de algún modo y se obtiene lo que se llama una *lógica multimodal resultante de la combinación*.

Normalmente se unen lógicas específicas, pues tendría poco sentido poner a trabajar juntas lógicas que tengan poder expresivo general.

Combinación de lógicas. Combinar lógicas es una técnica que está actualmente en estudio y expansión, inspirada principalmente en el interés por la modularidad. Permite definir sistemas formales altamente especializados. ¿Ensamblar lógicas nos ofrece algo nuevo? La respuesta es sí: no hay un estudio sistematizado de cómo combinar lógicas, tampoco hay un cuerpo establecido de resultados. Lo que sí existe es un núcleo de nociones y combinaciones exitosas que han surgido para una clase importante de lógicas.

Por un lado, como intuimos, existe un aspecto ingenieril o de diseño que nos lleva –como informáticos– a considerar a las lógicas pequeñas como bloques o unidades de manejo de conocimiento con los que podemos construir sistemas más grandes: podemos reutilizar los bloques, montar bloques unos con otros, sustituir un bloque por otro con iguales o mejores prestaciones. Por otro lado, debemos prestar atención al aspecto "lógico" de la combinación o montaje de bloques lógicos: como lo que estamos combinando son lógicas que poseen determinadas propiedades que seguramente consideramos ventajosas por algún motivo (como la decidibilidad, por ejemplo) pretendemos que las lógicas resultantes conserven las buenas propiedades de sus bloques componentes.

Los lógicos y los lógicos computacionales que se dedican a estos temas llaman a dicha cuestión, dice C. Areces, *el problema de transferencia*: sean Λ_1 y Λ_2 dos lógicas y sea P una propiedad que las lógicas puedan tener (como decidibilidad, f.m.p., alguna cota de complejidad, etc). Si $\oplus$ es un modo de combinar Λ_1 y Λ_2, ¿posee $\Lambda_1 \oplus \Lambda_2$ la propiedad P? Es importante resolver algunos problemas de transferencia al proponer una lógica combinada. Un primer principio relativo a transferencia parece indicar que si no hay interacción entre las lógicas (es decir, las lógicas no comparten símbolos excepto conectivos booleanos), la propiedad en cuestión se preserva. Pero aún en formas leves de interacción la transferencia de propiedades puede fallar.

A continuación presentamos sintaxis y semántica para un MAS diseñado como una combinación de lógicas de propósito determinado: ponemos a trabajar juntas las lógicas específicas que hemos presentado para creencias, objetivos, intención y acción. Luego de describir la combinación comentamos sobre la transferencia de propiedades en la lógica resultante.

Lenguaje de un sistema multiagente BDI. Al lenguaje modal básico presentado anteriormente le agregamos un conjunto finito de agentes A = {*x*, *y*, *z*...}. Las expresiones complejas son construidas del modo inductivo usual con los operadores lógicos clásicos y con los operadores unarios modales Bel_x, Int_x, $Goal_x$ y $Does_x$.

Axiomas "puente". Existen relaciones entre creencias, objetivos e intenciones de agentes, que describimos axiomáticamente como: $Goal_x \mathcal{A} \rightarrow Bel_x(Goal_x \mathcal{A})$ (introspección positiva de objetivos), $Int_x \mathcal{A} \rightarrow Bel_x(Int_x \mathcal{A})$ (introspección positiva de intenciones), $\neg Goal_x \mathcal{A} \rightarrow Bel_x(\neg Goal_x \mathcal{A})$ (introspección negativa de objetivos), $\neg Int_x \mathcal{A} \rightarrow Bel_x(\neg Int_x \mathcal{A})$ (introspección negativa de intenciones), e $Int_x \mathcal{A} \rightarrow Goal_x \mathcal{A}$ (intención implica objetivo).

Los axiomas que expresan interdependencias entre las creencias y las actitudes motivacionales (objetivos e intenciones) permiten ver que los agentes son conscientes de los objetivos e intenciones que tienen, así como de los que no tienen. Así como los hemos definido, los axiomas para actitudes motivacionales y sus aspectos combinados son mínimos en el sentido de que pretendemos manejarnos con condiciones necesarias y suficientes, tal como los definen B. Dunnin-Keplickz y R. Verbrugge. Notemos que no hay axiomas de "realismo fuerte" como puede considerarse a los axiomas $Goal_x \mathcal{A} \rightarrow Bel_x \mathcal{A}$ e $Int_x \mathcal{A} \rightarrow Bel_x \mathcal{A}$ que corresponderían, por ejemplo, a las ideas de que un agente cree que puede alcanzar sus objetivos e intenciones mediante la elección cuidadosa de sus acciones.

Tengamos en cuenta, de todos modos, que aspectos adicionales de creencias, deseos e intenciones siempre pueden modelarse agregando nuevos axiomas y creando extensiones más refinadas de estas lógicas mínimas.

Expresividad del sistema. Restricción. Solamente a los efectos de facilitar la presentación técnica de MAS como una combinación de lógicas, establecemos la siguiente restricción: una fórmula de la forma $Does_x \mathcal{A}$ siempre es aplicada a átomos que representan acciones simples (por ejemplo "comprar", "vender", "contestar"). Ejemplo: $Bel_x(Does_y$ 'Pagar') intuitivamente se lee "el agente *x* cree que el agente *y* paga". Con esta restricción los operadores modales normales interactúan con el operador Does de una manera limitada: no es posible escribir fórmulas como $Does_x(Does_y$ 'Pagar') o como $Does_x(Goal_y \mathcal{A})$ (que puede ser vista como una forma de persuasión: "el agente *x* hace que el agente *y* tenga a $\mathcal{A}$ como objetivo"). Esta restricción no impide que, en otras configuraciones diferentes para otros MAS algunos operadores puedan aparecer dentro del alcance de un Does.

Semántica del sistema multiagente. La estructura del sistema multiagente es una extensión de la definición de frame dada en la primera sección:

$$\mathcal{F} = (A, W, \{B_i\}_{i\in A}, \{G_i\}_{i\in A}, \{I_i\}_{i\in A}, \{D_i\}_{i\in A})$$

donde:

- A es un conjunto finito de agentes.
- W es un conjunto de situaciones, o mundos.
- $\{B_i\}_{i\in A}$ es un conjunto de relaciones de accesibilidad para los operadores de creencias Bel_x (hay un operador para cada agente, por lo tanto tenemos una relación de accesibilidad para cada uno de esos operadores). Las relaciones de accesibilidad B_i son transitivas (cumplen $\forall xyz\ (Rxy \wedge Ryz) \rightarrow Rxz$), euclideanas (cumplen $\forall xyz\ (Rxy \wedge Rxz) \rightarrow Ryz$) y seriales (sin límite a derecha, como las deónticas). Los frames con estas características son precisamente los que quedan deteminados por los axiomas que definen a la lógica de las creencias (por ello a la lógica de creencias se la llama de tipo KD45).

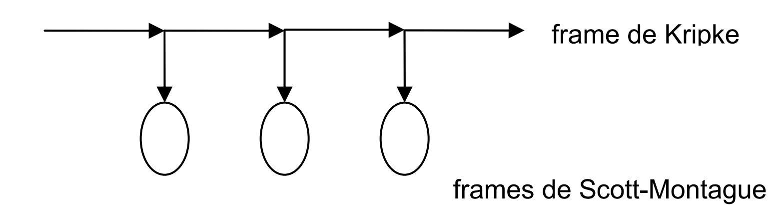
- $\{G_i\}_{i\in A}$ es el conjunto de relaciones de accesibilidad para cada uno de los operadores de objetivos $Goal_x$, cuya semántica es de necesidad estándar, K (definición 3.8).
- $\{I_i\}_{i\in A}$ es el conjunto de relaciones de accesibilidad respecto de los operadores de intenciones Int_x, relaciones que son seriales (la lógica de intenciones es KD, como la deóntica).
- $\{D_i\}_{i\in A}$ es una familia de conjuntos de relaciones de accesibilidad para los operadores Does, relaciones que son reflexivas, seriales, y satisfacen ciertas condiciones especiales de clausura (descriptas en *On the Axiomatisation of Elgesem´s Logic for Agency and Ability*, de G. Governatori y A. Rotolo).

Finalmente, definimos un *modelo* para nuestros sistemas multiagente como una estructura de la forma $\mathcal{M} = (\mathcal{F}, V)$ en que:

- $\mathcal{F}$ es un frame como definimos más arriba, y
- V es una función de valuación definida como sigue:
  i. condiciones booleanas estándar
  ii. $V(w, Bel_i\,\mathcal{A}) = true$ si y solo si $\forall v$ (si $B_i wv$ entonces $V(v,\mathcal{A}) = true$)
  iii. $V(w, Goal_i\,\mathcal{A}) = true$ si y solo si $\forall v$ (si $G_i wv$ entonces $V(v,\mathcal{A}) = true$)
  iv. $V(w, Int_i\,\mathcal{A}) = true$ si y solo si $\forall v$ (si $I_i wv$ entonces $V(v,\mathcal{A}) = true$)
  v. $V(w, Does_i\,\mathcal{A}) = true$ si y solo si $\exists\, \mathcal{D}_i \in D_i$ tal que $\forall v$ ($\mathcal{D}_i wv$ sii $V(v,\mathcal{A}) = true$)

V está definida como para los modelos de Kripke excepto para los operadores Does: la fórmula $Does_i\,\mathcal{A}$ es verdadera en w si y solo si existe un vecindario $\mathcal{D}_i$ de w, $\mathcal{D}_i \in D_i$ (con D_i conjunto de todos los vecindarios de w) en el que la fórmula $\mathcal{A}$ es verdadera.

Evaluación de fórmulas. Navegación dentro del frame. Notemos que es posible identificar dos "redes" de relaciones sobre W. La primera red, tal como está definido $\mathcal{F}$, corresponde al "cableado" de los operadores normales. La segunda red corresponde a las relaciones de accesibilidad para las modalidades Does. Podemos representar gráficamente a $\mathcal{F}$ como si hubiese un frame de Kripke "exterior", y frames de Scott-Montague "interiores":



Viéndolo de este modo, la intuición detrás de la evaluación de las fórmulas en el sistema multiagente es la siguiente: cuando *parseamos* una fórmula navegamos por el modelo de Kripke "exterior" hasta que una subfórmula que comienza con el operador Does aparece para ser

evaluada. Ahí nos "metemos" en un modelo de Scott-Montague. Evaluamos la subfórmula Does en el modelo de Scott-Montague y sustituimos en la fórmula original la subfórmula del Does con el resultado de esta evaluación (la sustitución la haremos con algún objeto que pertenezca al dominio de los modelos de Kripke tal como una variable proposicional o un valor de verdad) y continuamos con la evaluación que, de algún modo, en este punto, ha sido "homogeneizada".

Fibrado. El modo en el que hemos reorganizado la vista gráfica de nuestro MAS como un frame de Kripke "exterior" y frames de Scott-Montague "interiores" se corresponde con la técnica de combinación de lógicas llamada *fibrado*. La estructura fibrada resultante se define como una función fibrada $f_{\Lambda 1,\Lambda 2}$, una función total que asigna a cada elemento $w \in \mathcal{M}_{\Lambda 1}$ un modelo $\mathcal{M}_{\Lambda 2}$, con $\mathcal{M}_{\Lambda 1}$ y $\mathcal{M}_{\Lambda 2}$ modelos de las lógicas Λ_1 y Λ_2 respectivamente (la de Kripke y la de Scott-Montague). Así, una expresión del lenguaje $\mathcal{L}_2$ de la lógica Λ_2 es evaluada en $\mathcal{M}_{\Lambda 1}$ –donde es indefinida– a través de $f_{\Lambda 1,\Lambda 2}(w)$.

La restricción de expresividad que impusimos sobre el uso del Does (C. Smith y A. Rotolo) en este MAS favorece el fibrado presentado, ya que las lógicas fueron puestas a trabajar de una manera simple y de modo tal que, luego, el algoritmo de evaluación de fórmulas no resulta complejo: se trabaja en un modelo para lógicas normales, cuando se encuentra un Does se evalúa la subfórmula en un modelo no normal. Hemos dicho que es posible definir otros MAS donde los operadores modales sí pueden aparecer dentro del Does, por ejemplo donde sea posible escribir y evaluar fórmulas del tipo $\text{Does}_x(\text{Goal}_y\mathcal{A})$. En ellos se aplican otras técnicas de combinación de lógicas y otros algoritmos de evaluación de fórmulas.

Completitud. La lógica resultante de la combinación de lógicas específicas tal como fue presentada es completa. Para ello es suficiente construir un modelo canónico para dicha combinación y establecer que la lógica es fuertemente completa con respecto al modelo canónico. La prueba de completitud para la porción no normal de la lógica es, como imaginamos, intrincada, una muy clara exposición de la prueba de completitud de una lógica no normal Does aparece en la tesis de grado de F. Carbonari, publicada por la Universidad de La Plata.

Decidibilidad. Si las lógicas componentes son decidibles, es posible que la combinación resultante también lo sea. La lógica del Does posee la f.m.p., y las lógicas monomodales que usamos también. La lógica resultante de la combinación de las lógicas también es decidible. Dejamos al lector el armado de la prueba, que es larga pero no demasiado compleja.

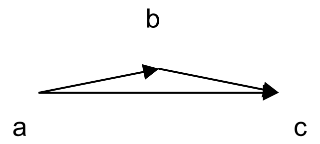
Complejidad. La lógica combinada presentada, a pesar de ser decidible, es EXPTIME completa (su problema de decisión tiene tiempo de ejecución exponencial, y otros problemas en la misma clase de problemas pueden reducirse al mismo). Existen algunas técnicas para reducir esta complejidad, como por ejemplo limitar la profundidad de fórmulas modales, poniendo un límite a la cantidad de operadores modales que pueden aparecer en las

subfórmulas de una fórmula. M. Dziubinski, R. Verbrugge y B. Dunnin-Keplicz han hecho algunas propuestas para el tratamiento de la complejidad en MAS.

Extensiones para la lógica combinada. Podemos extender sin mayores conflictos el MAS que hemos presentado con el operador deóntico de obligaciones O. También podríamos hacerlo con una lógica temporal. Por ejemplo, temporalizar la lógica presentada se reduce simplemente a montar sobre el MAS la maquinaria temporal con el mismo espíritu con el que visualizamos la maquinaria normal organizada sobre la no normal. Este fibrado ha sido descripto en un trabajo conjunto con A. Rotolo, A. Ambrosio y L. Mendoza, lo puntualizamos brevemente a continuación. Consideremos el modelo $(\mathcal{T}, <, g, t_0)$, entonces tenemos un frame "externo" $(\mathcal{T}, <)$ que se corresponde con la línea de evolución temporal, y t_0 es el instante inicial de tiempo. El sistema evoluciona en el sentido de que nuevos grupos, creencias, relaciones y obligaciones se van creando y también desarmando a lo largo del tiempo. Así, *g* es la función total que para cada punto t_i de la línea temporal "trae" un modelo $\mathcal{M}$ (como el definido más arriba) para evaluar.

Ejercicios

1. Sea el siguiente frame:



Se sabe que la fórmula p es verdadera en el mundo b y que la fórmula q es verdadera en los mundos a y c. Demostrar si:

i. $\Box p$ es verdadera en a
ii. $\Diamond p$ es verdadera en a
iii. $p \vee \Box q$ es verdadera en b
iv. $\Box q \rightarrow \Diamond p$ es verdadera en b

2. Probar que la fórmula $\Box p \rightarrow \Diamond p$ de la lógica modal no es válida en la clase de todos los frames.

3. Probar que la fórmula $(\Box p \wedge \Box q) \rightarrow \Box (p \wedge q)$ es válida en la clase de todos los frames.

4. Simbolizar utilizando los operadores deónticos O, P, F de obligación, permiso y prohibición según convenga. Algunas de las sentencias no tienen carácter deóntico, indicar cuáles.

i. El lector devolverá el libro en 15 días hábiles. Si el lector devuelve el libro en 15 días hábiles, no se le aplicará el apercibimiento administrativo del artículo 20. Si el lector no devuelve el libro en 15 días hábiles, se le aplicará el apercibimiento administrativo del artículo 20.

ii. Un círculo no puede ser cuadrado.

iii. Juan promete pagarle a Pedro $ 50.

iv. Juan firma: "Prometo pagarle a Pedro $ 50".

v. Juan se pone así mismo bajo la obligación de pagarle a Pedro $ 50.

vi. Juan está obligado.

vii. Juan debe pagar a Pedro $ 50.

viii. Fumar es perjudicial para la salud.

ix. Prohibido fumar.

x. Puede besar a la novia.

xi. Es obligatorio que lleves esta carta al correo. Por lo tanto, es obligatorio que o lleves esta carta al correo o la quemes.

xii. Tienes permitido o bien llevar la carta al correo o bien quemarla.

xiii. Debe haber paz en el mundo.

xiv. Está prohibido matar. Por lo tanto, están prohibidos ámbos matar y arrepentirse.

5. Suponer que la expresión $\Diamond p$ significa "p es tolerable".
   i. ¿Cuál es la lectura intuitiva del dual $\Box$ si definimos $\Box p \equiv \neg\Diamond\neg p$?
   ii. Se quiere formalizar un sistema moral (social/religioso/mafioso/de etiqueta y ceremonial, etc.) que captura esta interpretación de $\Diamond$ y $\Box$. Listar fórmulas que puedan considerarse principios lógicos para el sistema. Por ejemplo: $\Diamond p \vee \Diamond q \rightarrow \Diamond(p \vee q)$.
   iii. Simbolizar la proposición "si algo sucede, entonces es tolerable". ¿La incluiríamos como principio lógico en la lista previa? Fundamentar.
   iv. ¿Incluiríamos la fórmula $\Box(\Box p \rightarrow p) \rightarrow \Box p$ en la lista de principios? Fundamentar. ¿Cuál es su lectura intuitiva?
   v. ¿Y $p \rightarrow \Box\Diamond p$? Fundamentar. ¿Cuál es su lectura intuitiva?
   vi. ¿Cómo se modela en este sistema el comportamiento altruista?

6. Se quiere formalizar lógicamente la coexistencia de un sistema de la tolerancia caracterizado como en el inciso previo con los operadores $\Box$ y $\Diamond$ y un sistema de normas jurídicas caracterizado por los operadores O, P, y F, donde O tiene una semántica de necesidad, P de posibilidad y $Fp \equiv \neg Pp$. Simbolizar las proposiciones a continuación, y determinar (fundadamente) para cada una de ellas si constituyen o no principios lógicos de esta coexistencia de sistemas normativos.

i. Lo tolerable está permitido. (Pensar en el robo de la señal de cable.)
ii. Si algo está permitido, es tolerable. (Pensar en sistemas morales que, por ejemplo, legitiman el aborto, ¿qué pasa con los sectores que están bajo esas normas pero en contra de esa práctica?)
iii. Si algo está permitido, entonces es obligatorio tolerarlo. (Pensar en el derecho a huelga, manifestaciones, etc.)
iv. Si algo está prohibido, es obligatorio no tolerarlo. (Idem anterior.)

7. ¿Cómo se enuncian en lenguaje natural las siguientes proposiciones de la lógica combinada de normas/tolerancia del ítem previo? ¿Constituyen principios de la coexistencia de ambas lógicas? Fundamentar.
   i. $Op \rightarrow \Diamond p$
   ii. $\Box p \rightarrow \Box Op$
   iii. $Op \rightarrow O(\Diamond Op)$
   iv. $\neg(Pp \rightarrow \Box p)$

8. Formalizar las siguientes reglas de comportamiento con los operadores de obligación, permiso y prohibición O, P, y F.

De las conductas indecorosas en la mesa de mi señor:

(Texto anónimo, aunque atribuido a Leonardo Da Vinci, quien trabajó para los Médici, ca.1600)

Estos son (algunos de) los hábitos indecorosos que invitados a la mesa de mi señor no deben cultivar (y baso esto en mi observación de aquéllos que frecuentaron la mesa de mi señor durante el año pasado).

Ningún invitado ha de sentarse sobre la mesa, ni de espaldas a la mesa, ni sobre el regazo de cualquier otro invitado.

Tampoco ha de poner la pierna sobre la mesa.

Tampoco ha de sentarse bajo la mesa en ningún momento.

No ha de limpiar su armadura sobre la mesa.

No ha de tomar comida de la mesa y ponerla en su bolso o faltriquera para después comerla.

No ha de hacer figuras modeladas ni prender fuegos ni adiestrarse en hacer ruidos en la mesa (a menos que mi señor se lo pida).

No ha de tocar el laúd o cualquier otro instrumento que pueda ir en perjuicio de su vecino de mesa (a menos que mi señor se lo pida).

No ha de cantar, ni hacer discursos, ni vociferar improperios ni tampoco proponer acertijos obscenos si está sentado frente a una dama.

No ha de conspirar en la mesa (a menos que lo haga con mi señor).

Tampoco ha de prender fuego a su compañero mientras permanezca en la mesa.

No ha de golpear a los sirvientes (a menos que sea en defensa propia).

9. Algunas de las reglas previas de decoro en la mesa tienen contenido temporal. Identificarlas y modelarlas en el contexto de una lógica deóntica que se ha combinado con una temporal que tiene los operadores *P* y *F* para simbolizar "en el pasado" y "en el futuro".

10. Manejamos un lenguaje modal proposicional fundado sobre un conjunto finito A de agentes y un conjunto numerable de proposiciones, denotadas con p, q, r,… Expresiones complejas se forman sintácticamente a partir de ellas, en el modo inductivo usual, usando un operador $\bot$, el operador binario $\vee$, y modalidades unarias O y Does_x (donde el subíndice corre sobre el conjunto de agentes). Como el comportamiento proposicional de esta lógica es clásico, asumimos que $\top$, $\vee$, $\rightarrow$ se definen del modo usual. El operador Does debe entenderse para representar éxito en el actuar. En esta lógica combinada, formalizar algunas reglas de decoro en la mesa del ejercicio 8.

11. Se tiene una lógica multiagente que provee el operador Does. Para las siguientes fórmulas, dados dos agentes x e y cualesquiera, dar su lectura intuitiva:
    i. $\text{Does}_x \mathcal{A} \rightarrow (\text{Does}_x(\text{Does}_x \mathcal{A}))$
    ii. $(\text{Does}_y(\text{Does}_x \mathcal{A})) \rightarrow \text{Does}_y \mathcal{A}$

    Indicar si la última fórmula puede ser considerada un principio axiomático de una lógica de la acción. Fundamentar.

12. La noción de abstención dice que un agente se abstiene de hacer algo si y solo si puede hacerlo pero no lo hace. Definir la noción de abstención usando una lógica multiagente con los operadores deónticos usuales combinados con el operador de la acción $\text{Does}_x \mathcal{A}$.

13. Estudiar el impacto de los teoremas O$\top$ y $\text{Does}_x\top$ en las semánticas pretendidas para las lógicas deóntica y de la acción, respectivamente. Comparar los esquemas de axioma $\neg$O$\top$ de la lógica deóntica (que von Wright acepta en su sistema original) y $\neg\text{Does}_x \top$ de la lógica de la acción.

14. Probar que la fórmula de la lógica epistémica $(\neg K\mathcal{A} \rightarrow \neg K\mathcal{B}) \rightarrow (K\mathcal{B} \rightarrow K\mathcal{A})$ es una verdad lógica más allá de su contenido epistémico. Ofrecer su lectura intuitiva en lenguaje natural.

Bibliografía

Areces, C., Monz, C., de Nivelle, H., de Rijke, M. (1999). *The Guarded Fragment: Ins and Outs*. En J. Gerbrandy, M. Marx, M. de Rijke, and Y. Venema, editors, Essays Dedicated to Johan van Benthem on the Occasion of his 50th Birthday, Vossiuspers, AUP, Amsterdam. http://www.loria.fr/~areces/content/papers/files/j50b.pdf

Blackburn, P., de Rijke, M., Venema Y. (2001). *Modal Logic*. Cambridge University Press.

Carbonari, F. (2015). *Pruebas interesantes para una lógica multi-modal multi-agente: Un caso de estudio sobre Completitud.* Tesis de grado, Facultad de Informática UNLP.

Chellas, B. F. (1980). *Modal Logic. An Introduction.* Cambridge University Press.

Dunnin-Keplickz, B., Verbrugge, R. (2007). *Collective Intentions*. Fundamenta Informaticae. http://rinekeverbrugge.nl/PDF/Articles%20in%20refereed%20journal/Collectiveintentions-I02.pdf.

Dziubinski, M., Verbrugge, R. Dunnin-Keplicz, B. (2007). *Complexity issues in multiagent logics*. Fundamenta Informaticae, 75 (1-4):239-262.

Elgesem, D. (1997). *The Modal Logic of Agency*. Nordic Journal of Philosophical Logic. Vol 2 (2), 1-46. Scandinavian University Press. http://citeseerx.ist.psu.edu/viewdoc/download? doi=10.1.1.100.2796&rep=rep1& type=pdf

Governatori, G., Rotolo, A. (2005). *On the Axiomatisation of Elgesem´s Logic for Agency and Ability*. En Journal of Philosophical Logic. Volume 34, Issue 4, pp 403-431. http://web.stanford.edu/class/cs222/Governatori.pdf

Hamilton, A. G. (1981). *Lógica para Matemáticos.* (Traducción de M. R. Artalejo). Paraninfo, Madrid.

Hansson, B., Gardenfors, P. (1973). *A Guide to Intensional Semantics*. En Modality, Morality and Other Problems of Sense and Nonsense. Essays dedicated to Sören Hallden. CWK Gleerup, Lund, pp. 151-167.

Herrestad, H., Krogh, C. (1995). *Deontic Logic Relativised to Bearers and Counterparties*. En Bing, J. y Torvund, O. (editores) Anniversary Anthology in Computers and Law; COMPLEX - TANO, pp 453-522.

Jones, A. J. I. (1990). *Deontic logic and legal knowledge representation*. En Ratio Juris, 3:237-244.

Meyer, J.-J. Ch., Wieringa, R.J., Dignum, F.P.M. (1998). *The Role of Deontic Logic in the Specification of Information Systems*. En Logic for Databases and Information Systems. pp 71-115, Kluwer Academic Publishers Norwell, MA, USA. http://doc.utwente.nl/18384/1/role-deontic_meyer.pdf

Prolog. *Lenguaje y ambiente de programación en versión para educación e investigación*. http://www.swi-prolog.org/

Smith, C., Ambrossio, A., Mendoza, L., Rotolo, A. (2012). *Combinations of Normal and Non-normal Modal Logics for Modeling Collective Trust in Normative MAS*. En AI Approaches to the Complexity of Legal Systems. M. Palmirani (et al.) editores. LNAI 7639, pp. 189-203, Springer.

Smith, C., Rotolo, A. (2012). *Collective Trust and Normative Agents*. Logic Journal of the IGPL (2010) 18 (1): 195-213. Springer.

von Wright, G. H. (1951). *Deontic Logic*. Mind, LX (237), 1-15.

Wieringa, R. J., J-J. Ch. Meyer. (1994). *Applications of Deontic Logic in Computer Science: A Concise Overview*. En Deontic Logic in Computer Science, J-J. Ch. Meyer, R. J. Wieringa (editores). pp 17-40, John Wiley & Sons, Inc. New York, NY, USA. http://eprints.eemcs.utwente.nl/10663/01/applications-of-deontic-logic.pdf

CAPÍTULO 4
Lógica de programas

Ricardo Rosenfeld

Introducción

Estudiadas previamente las lógicas de proposiciones, de predicados y modal, en este último capítulo del libro nos basamos en ellas para tratar, de una manera bastante elemental, la verificación de programas. Introducimos la *verificación axiomática de programas*. Siguiendo el formato de los capítulos anteriores, desarrollamos los contenidos en el marco de lógicas esta vez de programas (o *teorías de programación*), y de este modo también consideramos fórmulas bien formadas, con sintaxis y semánticas determinadas, axiomas, reglas de inferencia, pruebas, nociones de satisfactibilidad, sensatez, completitud, etc.

La aproximación natural de la verificación de programas por la vía *operacional*, es decir *semántica*, resulta prohibitiva para programas complejos, sobre todo concurrentes, cuando hay numerosas computaciones y propiedades a probar. Una alternativa es, entonces, la verificación axiomática, *sintáctica*, muy difundida y cada vez con mayor soporte herramental, que plantea métodos de prueba con axiomas y reglas de inferencia asociados a las instrucciones de los lenguajes de programación, permitiendo probar la correctitud de un programa de la misma manera que se prueba un teorema. Las pruebas son guiadas en muchos casos por la estructura de los programas. Esta aproximación también puede emplearse como base para una metodología de desarrollo sistemático de programas, para construir un programa simultáneamente con su prueba de correctitud.

Estructuramos el capítulo en dos partes, una dedicada a la verificación de los programas de *entrada/salida*, y la otra a la verificación de los programas *reactivos*, en los que no hay una noción de terminación sino de interacción permanente con el entorno. A su vez, la verificación de los programas de entrada/salida se trata a lo largo de tres secciones, considerando distintos paradigmas, desde el más simple que es el *secuencial determinístico*, pasando por el *secuencial no determinístico*, y llegando al más complejo que es el *concurrente*.

En todos los casos trabajamos con lenguajes de programación muy representativos, los lenguajes *imperativos*. Por lo tanto tratamos con variables, instrucciones de asignación, secuencias de instrucciones, estados, etc. Para facilitar la exposición simplificamos notoriamente la complejidad de los lenguajes; por ejemplo, consideramos solamente variables de tipo entero y booleano, y no incluimos procedimientos. Además, para estudiar los distintos

paradigmas en un marco lo más unificado posible, los lenguajes de programación que utilizamos de la segunda sección en adelante son extensiones del que utilizamos en la primera.

Mientras que para la verificación de los programas de entrada/salida recurrimos a la lógica *clásica* de primer orden (segundo capítulo del libro), la verificación de los programas reactivos la desarrollamos utilizando lógica *temporal*, una de las instanciaciones posibles de la lógica modal (tercer capítulo), considerando sus fragmentos tanto proposicional como de primer orden, que para esta última familia de programas es más adecuada. En todos los casos, también para facilitar la exposición, trabajamos con el dominio semántico o interpretación de los *números enteros*.

En una serie de notas cerrando el capítulo (ver al final las Referencias y notas) profundizamos en distintos aspectos de lo tratado. Las dejamos para lo último para evitar desviar al lector del camino principal, bien introductorio, que nos hemos planteado.

Lógica de programas de entrada/salida

Empezamos el estudio de la verificación axiomática de programas con los programas de entrada/salida. Vamos a plantear distintas lógicas de programas, todas con fórmulas de la forma {p} S {q} y ⟨p⟩ S ⟨q⟩, siendo S un programa, escrito en un determinado lenguaje de programación, y el par (p, q) una *especificación* de S, siendo p la condición inicial de S, su *precondición*, y q la condición final de S, su *postcondición*. Las fórmulas se conocen como *fórmulas de correctitud*, y también como *ternas de Hoare* (C. Hoare fue el creador de esta aproximación). Las condiciones p y q son, a su vez, fórmulas de primer orden interpretadas en el dominio de los números enteros. Informalmente (precisamos en la siguiente sección), la fórmula {p} S {q} establece que si el programa S se ejecuta cuando se cumple la precondición p, y termina, entonces al final se cumple la postcondición q. Por ejemplo tenemos la siguiente fórmula:

{x = 0} x := x + 1 {x = 1}

En este caso se dice que el programa S es *parcialmente correcto* con respecto a la especificación (p, q). Por su parte, ⟨p⟩ S ⟨q⟩ establece que si S se ejecuta cuando se cumple p, entonces termina y al final se cumple q. En este otro caso se dice que el programa S es *totalmente correcto* con respecto a la especificación (p, q). Por ejemplo vale lo siguiente:

⟨x = 10⟩ while x ≠ 0 do x := x – 1 od ⟨x = 0⟩

Las expresiones *parcialmente correcto* y *totalmente correcto* provienen de la matemática, se relacionan con las funciones parciales y totales, definidas para algunos y todos los elementos de sus dominios, respectivamente. La idea así es asociar a un programa que no siempre

termina con una función parcial, y a un programa que siempre termina con una función total. Los dos casos de correctitud se consideran por separado porque para probarlos se debe recurrir a métodos de verificación distintos. Como la correctitud total implica la correctitud parcial, en la práctica esta separación se traduce en una prueba de correctitud parcial y una prueba de terminación, a partir de una misma precondición. Posteriormente, con ejemplos concretos, se aclara más este tema.

Las secciones que describen la verificación de programas considerando los distintos paradigmas referidos previamente, presentan básicamente la misma secuencia de presentación: (a) la sintaxis y semántica de las fórmulas de correctitud, (b) los métodos axiomáticos para probarlas y ejemplos sencillos de aplicación, y (c) comentarios con mayor o menor nivel de detalle sobre la *sensatez* y *completitud* de los métodos, propiedades ya referidas en los capítulos anteriores. Repasando, estas dos propiedades se definen de la siguiente manera. Un método es sensato si toda fórmula demostrada a partir de sus axiomas y reglas es verdadera (recordar que tratamos con la interpretación de los números enteros). Ya sabemos lo que significa que {p} S {q} sea verdadera: si S termina a partir de p, entonces al final se cumple q. También lo sabemos para el caso de ⟨p⟩ S ⟨q⟩. Como de costumbre, esta visión semántica la expresamos, respectivamente, con las notaciones:

$$\models \{p\}\ S\ \{q\} \quad y \quad \models \langle p\rangle\ S\ \langle q\rangle$$

así como las notaciones:

$$\vdash_{MA} \{p\}\ S\ \{q\} \quad y \quad \vdash_{MA^*} \langle p\rangle\ S\ \langle q\rangle$$

expresan que dichas fórmulas de correctitud se prueban mediante los métodos axiomáticos o sistemas deductivos MA y MA* (como estándar utilizamos al final del nombre de los métodos de prueba de correctitud total el símbolo *). El símbolo |= lo utilizamos sin subíndice porque hemos fijado con qué interpretación trabajaremos. Recíprocamente, un método es completo si toda fórmula verdadera puede ser demostrada a partir de sus axiomas y reglas. En síntesis, la sensatez de un método de prueba de correctitud parcial MA y de un método de prueba de correctitud total MA* establece, respectivamente, que para todo S, p, q:

$$\vdash_{MA} \{p\}\ S\ \{q\} \rightarrow\ \models \{p\}\ S\ \{q\} \quad y \quad \vdash_{MA^*} \langle p\rangle\ S\ \langle q\rangle \rightarrow\ \models \langle p\rangle\ S\ \langle q\rangle$$

Mientras que la completitud de un método de prueba de correctitud parcial MA y de un método de prueba de correctitud total MA* establece, respectivamente, que para todo S, p, q:

$$\models \{p\}\ S\ \{q\} \rightarrow\ \vdash_{MA} \{p\}\ S\ \{q\} \quad y \quad \models \langle p\rangle\ S\ \langle q\rangle \rightarrow\ \vdash_{MA^*} \langle p\rangle\ S\ \langle q\rangle$$

Notar que no se definen fórmulas de correctitud negadas, por lo que no se considera la posibilidad de que un método sea *inconsistente*, es decir que permita que se prueben fórmulas contradictorias.

Programas secuenciales determinísticos

Sintaxis de las fórmulas de correctitud

De acuerdo al esquema definido previamente, describimos primero la sintaxis de las fórmulas de correctitud {p} S {q} y ⟨p⟩ S ⟨q⟩ que vamos a considerar en esta sección.

Sintaxis del lenguaje de especificación

La sintaxis de las condiciones p y q, que conforman la especificación de un programa, ya la conocemos en general, corresponde a la de fórmulas de primer orden. En el caso genérico de una condición p, su definición inductiva es:

$p :: true \mid false \mid e_1 = e_2 \mid e_1 < e_2 \mid e_1 > e_2 \mid \ldots \mid \neg p \mid p_1 \vee p_2 \mid p_1 \wedge p_2 \mid \ldots \mid \exists x{:}p \mid \forall x{:}p$

$e :: m \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \, . \, e_2 \mid \ldots \mid$ if B then e_1 else e_2 fi

$B :: true \mid false \mid e_1 = e_2 \mid e_1 < e_2 \mid e_1 > e_2 \mid \ldots \mid \neg B \mid B_1 \vee B_2 \mid B_1 \wedge B_2 \mid \ldots$

e y B son expresiones enteras y booleanas, respectivamente, definidas inductivamente de esta manera. Las expresiones enteras se construyen a partir de constantes m y variables x. true y false son las constantes booleanas. La expresión entera if B then e_1 else e_2 fi no es habitual; su semántica informal (después precisamos) establece que si B es verdadera entonces el valor de la expresión es el de e_1, y si B es falsa entonces el valor es el de e_2.

Sintaxis del lenguaje de programación

Los programas, por su parte, pertenecen al lenguaje de programación PLW (por *programming lenguaje while* en inglés, es decir lenguaje de programación con while). Es un lenguaje secuencial determinístico muy simple del tipo *Algol* o *Pascal*, como se lo suele referenciar. La sintaxis de un programa S de PLW se define inductivamente de la siguiente manera:

S :: skip | x := e | S_1 ; S_2 | if B then S_1 else S_2 fi | while B do S_1 od

Notar que PLW no incluye variables de tipo booleano. La semántica de PLW es la habitual (la definición formal la presentamos enseguida):

- La instrucción *skip* es atómica (se consume en un paso), y no tiene ningún efecto sobre las variables. Se puede usar, por ejemplo, para escribir una selección condicional sin instrucción else.

- La *asignación* x := e también se considera atómica para simplificar, y asigna el valor de *e* a la variable x.
- La *secuencia* S_1 ; S_2, ejecuta S_1 y luego S_2.
- La *selección condicional* if B then S_1 else S_2 fi, ejecuta S_1 si B es verdadera o S_2 si B es falsa.
- Finalmente la *repetición* while B do S od, ejecuta S mientras B sea verdadera (primero evalúa B y luego si corresponde ejecuta S). Termina cuando B es falsa.

Por ejemplo, el siguiente programa PLW calcula mediante restas sucesivas la división entera entre $x \geq 0$ e $y > 0$, obteniendo el cociente en c y el resto en r:

$$S_{div} :: \; c := 0 \; ; \; r := x \; ; \; \text{while } r \geq y \text{ do } r := r - y \; ; \; c := c + 1 \text{ od}$$

Introducción a la semántica de las fórmulas de correctitud

Al igual que para la sintaxis, para definir ahora la semántica de las fórmulas {p} S {q} y ⟨p⟩ S ⟨q⟩ tenemos que definir lo correspondiente al lenguaje de especificación y al lenguaje de programación. Para ello, debemos formalizar primero la noción de *estado*.

A diferencia de las constantes, los valores de las variables de un programa PLW no son fijos sino que se asignan a partir de estados (modificados por las instrucciones de asignación). Un estado es una función σ que asigna a toda variable un valor de su tipo, en este caso un número entero. Puede verse como una "instantánea", en un momento determinado, de los contenidos de las variables de un programa. Se corresponde, en un sentido, con el concepto lógico de *valoración en una interpretación*. La expresión σ(x) denota el contenido de la variable x según el estado σ, y el conjunto de todos los estados se denota con Σ. Para denotar que una variable x tiene un valor particular m en un estado σ, se utiliza la expresión σ[x|m], lo que se conoce como *variante de un estado*, útil para definir después la semántica de la instrucción de asignación. Formalmente, dadas x e y, se define: σ[x|m](y) = m si se cumple x = y, o bien σ(y) si se cumple x ≠ y.

De esta manera, la visión semántica de una precondición p para un programa S es la de un conjunto inicial de estados: solo a partir de todos y cada uno de ellos interesa conocer cómo se comporta S. De manera análoga, la postcondición q denota el conjunto de estados finales posibles. Por ejemplo, tomando el programa S_{div} anterior, la fórmula:

$$\langle x \geq 0 \wedge y > 0 \rangle \; S_{div} \; \langle x = y \,.\, c + r \wedge r < y \wedge r \geq 0 \rangle$$

establece que a partir de todo estado inicial en el que se cumple $x \geq 0$ e $y > 0$ (el resto de las variables puede tener cualquier valor), el programa S_{div} termina y lo hace en un estado final en el que x vale $y \,.\, c + r$, y r cumple $r < y \wedge r \geq 0$. La especificación $(x \geq 0 \wedge y > 0,\; x = y \,.\, c + r \wedge r < y \wedge r \geq 0)$ establece así la relación que debe existir entre los estados iniciales y los estados finales del programa S_{div}.

Como las pre y postcondiciones formulan aserciones o enunciados sobre estados, se las conoce también justamente como *aserciones*. De esta manera vamos a denominar de ahora en más Assn (por *assertions* en inglés) al lenguaje de especificación.

Semántica del lenguaje de especificación

Para definir la semántica de Assn, familiar para nosotros por lo visto en un capítulo anterior, utilizamos las siguientes funciones semánticas:

$T : Assn \rightarrow (\Sigma \rightarrow \{verdadero, falso\})$

$V : Iexp \rightarrow (\Sigma \rightarrow Z)$

$W : Bexp \rightarrow (\Sigma \rightarrow \{verdadero, falso\})$

$+ : Z \times Z \rightarrow Z$ (de manera similar se definen – , **.** , etc.)

$= : Z \times Z \rightarrow \{verdadero, falso\}$ (de manera similar se definen < , > , etc.)

$\neg : \{verdadero, falso\} \rightarrow \{verdadero, falso\}$ (de manera similar se definen $\vee$, $\wedge$, etc.)

Z es el conjunto de los números enteros, Iexp el conjunto de las expresiones enteras de PLW (por *integer expressions* en inglés), y Bexp el conjunto de las expresiones booleanas de PLW (por *boolean expressions* en inglés). Con estas funciones, de manera inductiva podemos asociar a toda aserción y expresión, estado mediante, su correlato semántico. Como para describir la función semántica T de las aserciones se debe recurrir a la descripción de las funciones semánticas V y W de las expresiones enteras y booleanas, respectivamente, en lo que sigue habremos logrado no solo definir la semántica de las aserciones sino también la semántica de las expresiones enteras y booleanas del lenguaje PLW. Definimos:

- $V(m)(\sigma) = m$ (a la constante m se le asocia el número entero m)
- $V(x)(\sigma) = \sigma(x)$
- $V(e_1 + e_2)(\sigma) = V(e_1)(\sigma) + V(e_2)(\sigma)$ (de manera similar se definen – , **.** , etc.)
- $V(\text{if } B \text{ then } e_1 \text{ else } e_2 \text{ fi})(\sigma) = \text{if } W(B)(\sigma) \text{ then } V(e_1)(\sigma) \text{ else } V(e_2)(\sigma) \text{ fi}$
- W(true)(σ) = verdadero, y W(false)(σ) = falso
- $W(e_1 = e_2)(\sigma) = (V(e_1)(\sigma) = V(e_2)(\sigma))$ (de manera similar se definen < , > , etc.)
- $W(\neg B)(\sigma) = \neg W(B)(\sigma)$ (de manera similar se definen $\vee$, $\wedge$, etc.)
- T(true)(σ) = verdadero, y T(false)(σ) = falso
- $T(e_1 = e_2)(\sigma) = (V(e_1)(\sigma) = V(e_2)(\sigma))$ (de manera similar se definen < , > , etc.)
- $T(\neg p)(\sigma) = \neg T(p)(\sigma)$ (de manera similar se definen $\vee$, $\wedge$, etc.)
- $T(\exists x{:}p)(\sigma)$ = verdadero si y solo si existe m tal que $T(p)(\sigma[x|m])$ = verdadero
- $T(\forall x{:}p)(\sigma)$ = verdadero si y solo si todo m cumple que $T(p)(\sigma[x|m])$ = verdadero

Si se cumple T(p)(σ) = verdadero, diremos que el estado σ *satisface* la aserción p, lo que también se puede expresar con σ |= p. Del mismo modo podemos utilizar σ |= B para el caso

de las expresiones booleanas. $\sigma \not\models p$ abrevia $\neg(\sigma \models p)$, y $\sigma(e)$ y $\sigma(B)$ abrevian $V(e)(\sigma)$ y $W(B)(\sigma)$, respectivamente. Otra convención es denotar con true al conjunto de todos los estados y con false al conjunto vacío de estados, por lo que para todo estado σ se define $\sigma \models$ true, y $\sigma \not\models$ false.

Semántica del lenguaje de programación

La semántica del lenguaje de especificación Assn y de los sublenguajes de las expresiones enteras y booleanas de PLW se definió *denotacionalmente*. Mediante distintas funciones se determinó el valor de las construcciones de los lenguajes a partir del valor de sus componentes (la idea de construcciones lingüísticas que denotan valores da el nombre a este tipo de semántica). La semántica de las instrucciones de PLW la vamos a definir en cambio *operacionalmente*. Esta modalidad está ampliamente difundida, fundamentalmente porque para lenguajes complejos el uso de la semántica denotacional se torna muy dificultoso. Las descripciones se hacen en términos de las operaciones de una máquina abstracta, estableciendo cómo un programa transforma estados a partir de un estado inicial.

Más precisamente, dados un programa S y un estado inicial σ, se asocia a la *configuración inicial* $C_0 = (S, \sigma)$ una *computación* $\pi(S, \sigma)$, que es una secuencia de configuraciones $C_0 \rightarrow C_1 \rightarrow \ldots$, donde cada C_i es un par (S_i, σ_i), siendo S_i la *continuación sintáctica* (lo que le falta al programa para terminar), y σ_i el estado corriente. De esta manera, una computación no puede ser extendida, es *maximal*. Las computaciones $\pi(S, \sigma)$ se definen inductivamente por medio de una *relación de transición* entre configuraciones, basada en la sintaxis de S. Las computaciones son finitas o infinitas. Una computación finita $C_0 \rightarrow \ldots \rightarrow C_k$ se abrevia con $C_0 \rightarrow^* C_k$. C_k se denomina *configuración terminal*, y es un par (E, σ_k). E (por *empty*, vacío en inglés) es la *continuación sintáctica vacía*, no forma parte del lenguaje, solo se utiliza para indicar que el programa asociado terminó, y cumple que S ; E = E ; S = S para todo S. El estado final de $\pi(S, \sigma)$ se denota con $val(\pi(S, \sigma))$. Si $\pi(S, \sigma)$ es infinita, se escribe $val(\pi(S, \sigma)) = \perp$, y se define que $val(\pi(S ; S', \sigma)) = \perp$ para todo S' (es decir que la no terminación se propaga). Al símbolo ⊥ se lo conoce como *estado indefinido*, para diferenciarlo de los que están definidos, que se conocen como *estados propios*. Además de la expresión $val(\pi(S, \sigma))$ también se utiliza $M(S)(\sigma)$, siendo $M: PLW \rightarrow (\Sigma \rightarrow \Sigma)$ la función semántica asociada al lenguaje PLW. Dado S, M(S) es una función total, porque cuando $\pi(S, \sigma)$ no termina se cumple $M(S)(\sigma) = \perp$, y además $M(S)(\perp) = \perp$. Para que las funciones semánticas V, W y T sean totales, también deben completarse sus definiciones teniendo en cuenta el estado indefinido. Naturalmente, para toda aserción p se cumple $\perp \not\models p$.

La semántica de las instrucciones de PLW se establece definiendo inductivamente una relación de transición → entre configuraciones, de la siguiente manera:

- $(skip, \sigma) \rightarrow (E, \sigma)$
- $(x := e, \sigma) \rightarrow (E, \sigma[x|\sigma(e)])$ (la expresión $\sigma[x|\sigma(e)]$ se puede abreviar con $\sigma[x|e]$)
- Si $(S, \sigma) \rightarrow (S', \sigma')$, entonces para todo T: $(S ; T, \sigma) \rightarrow (S' ; T, \sigma')$

- Si $\sigma(B)$ = verdadero, entonces (if B then S_1 else S_2 fi, σ) $\rightarrow$ (S_1, σ)
  Si $\sigma(B)$ = falso, entonces (if B then S_1 else S_2 fi, σ) $\rightarrow$ (S_2, σ)
- Si $\sigma(B)$ = verdadero, entonces (while B do S od, σ) $\rightarrow$ (S ; while B do S od, σ)
  Si $\sigma(B)$ = falso, entonces (while B do S od, σ) $\rightarrow$ (E, σ)

De la definición anterior se infiere el determinismo de PLW: un programa tiene una sola computación, solo una configuración sucede a otra. Notar que no se explicitan las distintas formas que pueden adoptar las computaciones; de todos modos se pueden desarrollar fácilmente (queda como ejercicio para el lector). A diferencia de la instrucción de asignación, el skip y la evaluación de una expresión booleana no modifican el estado corriente, y todas ellas se consumen en un solo paso. Notar también que como para todo S se cumple S ; E = E ; S = S, si $val(\pi(S, \sigma)) = \sigma' \neq \perp$ entonces (S ; T, σ) $\rightarrow^*$ (T, σ').

Para facilitar la exposición en esta sección, posponemos para la siguiente la posibilidad de *falla* de un programa, es decir que termine incorrectamente (podemos asumir entre otras cosas en PLW, por ejemplo, que dividir por cero da cero). Contemplar esta posibilidad amerita utilizar un estado de falla f, y definir que si se cumple $val(\pi(S, \sigma)) = f$, entonces $val(\pi(S ; S', \sigma)) = f$ para todo S', es decir que la falla se propaga, tal como definimos en el caso de la no terminación. Naturalmente, para toda aserción p se cumple $f \not\models p$.

Precisando la semántica de las fórmulas de correctitud

Con las definiciones previas ahora sí podemos precisar la semántica de las fórmulas de correctitud parcial {p} S {q} y de correctitud total $\langle p \rangle$ S $\langle q \rangle$, es decir qué significa formalmente que un programa S sea parcial o totalmente correcto con respecto a una especificación (p, q), respectivamente. El primer caso se cumple si y solo si para todo estado σ:

$$(\sigma \models p \ \wedge \ val(\pi(S, \sigma)) \neq \perp) \rightarrow val(\pi(S, \sigma)) \models q$$

Y el segundo caso se cumple si y solo si para todo estado σ:

$$\sigma \models p \rightarrow (val(\pi(S, \sigma)) \neq \perp \ \wedge \ val(\pi(S, \sigma)) \models q)$$

Las pruebas de correctitud parcial y total requieren técnicas distintas, lo que justifica la separación de los dos criterios. Ya sugerido antes (la prueba queda como ejercicio para el lector), la correctitud total de S con respecto a (p, q) se puede expresar de la siguiente manera:

$$\{p\}\ S\ \{q\} \ \wedge \ \langle p \rangle\ S\ \langle true \rangle$$

Así, un camino natural para verificar que un programa S es totalmente correcto con respecto a una especificación (p, q) es: (a) probar que ejecutado a partir de la precondición p, si S termina, al final se cumple la postcondición q, y (b) probar que ejecutado a partir de la

precondición p, S termina. En la práctica, en (b) se deben considerar solamente todos y cada uno de los while de S, porque son la única fuente de no terminación. Este camino de dos pruebas, de correctitud parcial y terminación, es el que describimos en lo que sigue mediante los métodos axiomáticos respectivos H y H* (H en homenaje a Hoare).

Método H de verificación de correctitud parcial

Los axiomas y reglas del método H son:

- Axioma del skip (SKIP) $\{p\}\ \text{skip}\ \{p\}$
- Axioma de la asignación (ASI) $\{p[x|e]\}\ x := e\ \{p\}$
- Regla de la secuencia (SEC)

$$\frac{\{p\}\ S_1\ \{r\}\ ,\ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1\ ;\ S_2\ \{q\}}$$

- Regla del condicional (COND)

$$\frac{\{p \wedge B\}\ S_1\ \{q\}\ ,\ \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}\ \{q\}}$$

- Regla de la repetición (REP)

$$\frac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \text{while } B \text{ do } S \text{ od}\ \{p \wedge \neg B\}}$$

- Regla de consecuencia (CONS)

$$\frac{p \rightarrow p_1\ ,\ \{p_1\}\ S\ \{q_1\}\ ,\ q_1 \rightarrow q}{\{p\}\ S\ \{q\}}$$

Los axiomas se corresponden con las instrucciones atómicas de PLW, y las reglas con las instrucciones compuestas, salvo el caso especial de la regla CONS que explicamos después.
El axioma SKIP establece que si una aserción se cumple antes de la ejecución de un skip, sigue valiendo después.

El axioma ASI establece que si se cumple p en términos de x después de la ejecución de una asignación x := e, significa que antes de la ejecución se cumple p en términos de *e*. La expresión p[x|e] denota la sustitución en la aserción p de todas las ocurrencias libres de la variable x por la expresión *e*. Por ejemplo, con ASI se puede probar:

$$\{x + 1 \geq 0\}\ x := x + 1\ \{x \geq 0\}$$

Este axioma se lee “hacia atrás”, de derecha a izquierda, lo que impone una forma de desarrollar las pruebas de H en el mismo sentido, de la postcondición a la precondición. Si bien resulta más natural plantear un axioma que se lea “hacia adelante”, como:

$$\{\text{true}\}\ x := e\ \{x = e\}$$

este esquema no sirve, es falso cuando la expresión *e* incluye la variable x. Por ejemplo, si e = x + 1, se obtendría la fórmula falsa:

$$\{true\}\ x := x + 1\ \{x = x + 1\}$$

El problema radica en que la x de la parte derecha de la postcondición se refiere a la variable antes de la asignación, mientras que la x de la parte izquierda se refiere a la variable luego de la asignación. Una forma correcta de ASI “hacia adelante” es:

$$\{p\}\ x := e\ \{\exists z: p[x|z] \wedge x = e[x|z]\}$$

tal que e[x|z] denota la sustitución en la expresión *e* de las x por las z, pero nos quedaremos con la forma “hacia atrás” porque es más simple y es la más difundida.

La regla SEC establece que del cumplimiento de $\{p\}\ S_1\ \{r\}$ y $\{r\}\ S_2\ \{q\}$ se deriva el cumplimiento de $\{p\}\ S_1 ; S_2\ \{q\}$. La aserción r actúa como nexo para probar la secuencia de S_1 con S_2 y luego se descarta, es decir que no se propaga a lo largo de las pruebas. Hay una forma más general de la regla que es la siguiente:

$$\frac{\{p\}\ S_1\ \{r_1\}\ ,\ \{r_1\}\ S_2\ \{r_2\}\ ,\ \dots\ ,\ \{r_{n-1}\}\ S_n\ \{q\}}{\{p\}\ S_1\ ;\ S_2\ ;\ \dots\ ;\ S_n\ \{q\}}$$

Este esquema se puede obtener del original. De todos modos lo incluimos en H para acortar las pruebas.

La regla COND impone una manera de verificar una selección condicional estableciendo un único punto de entrada y un único punto de salida, correspondientes a la precondición p y la postcondición q, respectivamente. A partir de p, se cumpla o no la condición B, debe darse que luego de la ejecución de S_1 o S_2 se cumple q.

La regla REP se centra en una aserción invariante p, que debe cumplirse al comienzo de un while y luego de toda iteración del mismo. Mientras valga la condición B del while, la ejecución del cuerpo S debe preservar p, y por eso al terminar la instrucción (si termina) se cumple $p \wedge \neg B$. Claramente REP no asegura la terminación, y su forma muestra que la correctitud parcial puede probarse por inducción. Efectivamente, es esencialmente con la regla REP que se manifiesta el carácter inductivo del método H para verificar la correctitud parcial de los programas PLW. La técnica inductiva asociada se conoce como *computacional*, porque consiste en probar por inducción que una propiedad se cumple a lo largo de una computación.

Finalmente, la regla CONS permite reforzar las precondiciones y debilitar las postcondiciones de las fórmulas de correctitud. Por ejemplo, si $r \rightarrow p$ y $\{p\}\ S\ \{q\}$, entonces por CONS se prueba $\{r\}\ S\ \{q\}$. En particular, se pueden modificar fórmulas con aserciones equivalentes. Notar que por el uso de esta regla, algunos pasos de una prueba en H pueden contener directamente aserciones. CONS es una regla especial, no depende de PLW sino del dominio semántico, en este caso los números enteros. Es una regla semántica más que sintáctica. Actúa como interface entre las fórmulas de correctitud y los enunciados verdaderos de los enteros. Sin esta regla el método H no sería un método completo (*relativamente*

completo como veremos enseguida). Llamando Tr (por *true*, verdadero) al conjunto de todos los enunciados verdaderos de los enteros, una prueba en H debe entenderse entonces como:

$$Tr \vdash_H \{p\}\ S\ \{q\}$$

es decir que al método H se le agregan como axiomas todos los enunciados verdaderos de los enteros. En contraposición con lo usual en los métodos axiomáticos, el conjunto Tr no es *recursivo*, ni siquiera *recursivamente numerable*. Es que el camino natural de extender H con una axiomática de los números enteros no alcanza para obtener la completitud, debido al *teorema de incompletitud de Gödel*. De modo tal que la completitud es relativa al conjunto de los enunciados verdaderos de los enteros. La ventaja de definir H de esta manera es que se separan claramente los elementos dependientes de los programas PLW de los elementos dependientes de los datos, para poner foco en los primeros.

Además de la regla CONS, existe otra regla universal aplicable en todo método de verificación, aunque no se la suele incluir explícitamente. Es la regla de instanciación (INST), que permite instanciar las pre y postcondiciones de las fórmulas de correctitud a valores específicos del dominio semántico, en este caso los números enteros. Su uso está ligado al empleo de variables de *especificación* (también llamadas variables *lógicas*), las cuales no forman parte de los programas y sirven para "congelar" los contenidos de las variables de programa, que naturalmente pueden cambiar desde el comienzo hasta el final. Por ejemplo, la especificación (x = X, x = X + 1) es satisfecha por un programa que incrementa en 1 su entrada x, siendo X una variable de especificación empleada para guardar el valor inicial de la variable de programa x. Las variables de especificación no pueden ser accedidas ni modificadas, y están implícitamente cuantificadas universalmente. La forma de la regla INST es:

$$\frac{f\ (X)}{f\ (e)}$$

tal que f es una fórmula de correctitud que incluye una variable de especificación X, y *e* es una expresión entera. Si X ocurre libre en la precondición y la postcondición, *e* no debe incluir variables de programa, para asegurar que se instancie siempre el mismo valor. Por ejemplo, (x = X, x = X + 1) no se puede instanciar con e = x, porque ningún programa satisface (x = x, x = x + 1); sucede que los valores de x a izquierda y derecha de la especificación son distintos.

Por la completitud del método H, agregarle axiomas y reglas es redundante. De todos modos esta práctica es usual para acortar las pruebas. Los siguientes son algunos ejemplos de axiomas y reglas auxiliares bastante comunes que se le agregan a H:

- Axioma de invariancia (INV) {p} S {p}
  cuando ninguna variable libre de la aserción p es modificable por el programa S

- Regla de la disyunción (OR)

$$\frac{\{p\}\ S\ \{q\}\ ,\ \{r\}\ S\ \{q\}}{\{p \lor r\}\ S\ \{q\}}$$

- Regla de la conjunción (AND)

$$\frac{\{p_1\}\ S\ \{q_1\}\ ,\ \{p_2\}\ S\ \{q_2\}}{\{p_1 \land p_2\}\ S\ \{q_1 \land q_2\}}$$

El axioma INV suele emplearse en combinación con la regla AND. De hecho se suele utilizar directamente la siguiente regla, denominada justamente regla de invariancia:

$$\frac{\{p\}\ S\ \{q\}}{\{r \land p\}\ S\ \{r \land q\}}$$

cuando ninguna variable libre de r es modificable por S.

La regla OR es útil para probar una fórmula de correctitud cuya precondición es la disyunción de dos aserciones (se puede generalizar a más aserciones): en lugar de propagar a lo largo de una prueba información necesaria para establecer oportunamente la disyunción, se puede recurrir a esta regla, que permite una prueba por casos. Una forma particular de la regla OR de uso habitual es:

$$\frac{\{p \land r\}\ S\ \{q\}\ ,\ \{p \land \neg r\}\ S\ \{q\}}{\{p\}\ S\ \{q\}}$$

Esta regla es útil cuando la prueba de {p} S {q} se facilita reforzando la precondición con dos aserciones complementarias.

La regla AND sirve para probar una fórmula de correctitud tal que su precondición y su postcondición son conjunciones de dos aserciones (se puede generalizar a más aserciones). Se recurre a esta regla para lograr una prueba incremental, y así evitar propagar información necesaria para establecer oportunamente las conjunciones.

Antes de desarrollar un ejemplo de aplicación de H, cabe destacar una propiedad del método que es su *composicionalidad*. En efecto, H adhiere al principio de que si S está compuesto por subprogramas S_1, S_2, …, S_n, entonces que se cumpla {p} S {q} depende solamente de que se cumplan determinadas fórmulas $\{p_1\}\ S_1\ \{q_1\}$, $\{p_2\}\ S_2\ \{q_2\}$, …, $\{p_n\}\ S_n\ \{q_n\}$, sin necesidad de referencia alguna a la estructura interna de los S_i. Esta propiedad permite entonces verificar (y construir) modularmente un programa, favoreciendo la escalabilidad del método. La composicionalidad se cumple también en el método H* y en los métodos del paradigma secuencial no determinístico. En cambio se dificulta conservarla cuando se trata con programas concurrentes, como veremos más adelante.

Ejemplo de aplicación del método H

Vamos a probar la correctitud parcial del programa de división entera presentado antes, es decir la fórmula:

$$\{x \geq 0 \wedge y > 0\}\ S_{div}\ \{x = y \,.\, c + r \wedge r < y \wedge r \geq 0\}$$

siendo el programa S_{div}:

$$S_{div} :: \ c := 0 \ ; \ r := x \ ; \ \text{while } r \geq y \text{ do } r := r - y \ ; \ c := c + 1 \text{ od}$$

La dificultad de la verificación de un programa radica fundamentalmente en encontrar aserciones en distintas locaciones, en particular los invariantes de los while, para asociarlas con las premisas de alguna regla y obtener la conclusión deseada (siempre en el marco de esta exposición que se focaliza en la verificación de programas, porque la buena práctica es construir un programa en simultáneo con su verificación).

Proponemos como invariante del while la aserción $p = (x = y \,.\, c + r \wedge r \geq 0)$, obtenida por una técnica habitual que es la generalización de la postcondición del while, en este caso $x = y \,.\, c + r \wedge r < y \wedge r \geq 0$. Cuando el programa termina se cumple $r < y$, y de la conjunción de esta condición y el invariante se alcanza la postcondición buscada. Podemos estructurar la prueba de la siguiente manera:

a. $\{x \geq 0 \wedge y > 0\}\ c := 0 \ ; \ r := x\ \{p\}$. A partir de la precondición del programa, las asignaciones iniciales conducen al cumplimiento por primera vez del invariante p.
b. $\{p\}\ \text{while } r \geq y \text{ do } r := r - y \ ; \ c := c + 1 \text{ od}\ \{p \wedge \neg(r \geq y)\}$. El cuerpo del while preserva el invariante p, y por eso si el while termina se alcanza $p \wedge \neg(r \geq y)$.
c. $(p \wedge \neg(r \geq y)) \rightarrow x = y \,.\, c + r \wedge r < y \wedge r \geq 0$. La postcondición del while implica la postcondición del programa.

Prueba de a.

1. $\{x = y \,.\, c + x \wedge x \geq 0\}\ r := x\ \{p\}$ (ASI)
2. $\{x = y \,.\, 0 + x \wedge x \geq 0\}\ c := 0\ \{x = y \,.\, c + x \wedge x \geq 0\}$ (ASI)
3. $\{x = y \,.\, 0 + x \wedge x \geq 0\}\ c := 0 \ ; \ r := x\ \{p\}$ (1, 2, SEC)
4. $(x \geq 0 \wedge y > 0) \rightarrow (x = y \,.\, 0 + x \wedge x \geq 0)$ (MAT)
5. $\{x \geq 0 \wedge y > 0\}\ c := 0 \ ; \ r := x\ \{p\}$ (3, 4, CONS)

En el paso 4 se recurre a un enunciado verdadero de los números enteros, por eso se indica MAT, por matemática (podemos verlo como la invocación a un oráculo que nos devuelve el enunciado que necesitamos).

Prueba de b.

6. $\{x = y \,.\, (c + 1) + r \wedge r \geq 0\}\ c := c + 1\ \{p\}$ (ASI)
7. $\{x = y \,.\, (c + 1) + (r - y) \wedge r - y \geq 0\}\ r := r - y\ \{x = y \,.\, (c + 1) + r \wedge r \geq 0\}$ (ASI)

8. $\{x = y \cdot (c + 1) + (r - y) \wedge r - y \geq 0\}$ $r := r - y$; $c := c + 1$ $\{p\}$ (6, 7, SEC)

9. $(p \wedge r \geq y) \rightarrow (x = y \cdot (c + 1) + (r - y) \wedge r - y \geq 0)$ (MAT)

10. $\{p \wedge r \geq y\}$ $r := r - y$; $c := c + 1$ $\{p\}$ (8, 9, CONS)

11. $\{p\}$ while $r \geq y$ do $r := r - y$; $c := c + 1$ od $\{p \wedge \neg(r \geq y)\}$ (10, REP)

Prueba de c.

12. $(p \wedge \neg(r \geq y)) \rightarrow (x = y \cdot c + r \wedge r < y \wedge r \geq 0)$ (MAT)

Los pasos finales de la prueba son:

13. $\{x \geq 0 \wedge y > 0\}$ S_{div} $\{p \wedge \neg(r \geq y)\}$ (5, 11, SEC)

14. $\{x \geq 0 \wedge y > 0\}$ S_{div} $\{x = y \cdot c + r \wedge r < y \wedge r \geq 0\}$ (12, 13, CONS)

También se puede probar $\{x \geq 0 \wedge y \geq 0\}$ S_{div} $\{x = y \cdot c + r \wedge r < y \wedge r \geq 0\}$, es decir permitiendo que el divisor *y* sea 0 (queda como ejercicio para el lector).

Método H* de verificación de correctitud total

El método H* difiere de H solo en la regla de la repetición, porque el while es la única instrucción que puede provocar no terminación. Como convención de notación, a los nombres de los axiomas y reglas de H* se les agrega al final el símbolo *, y las pre y postcondiciones se delimitan con ⟨ ⟩.

La única novedad a presentar es, entonces, la regla REP*. REP* mantiene naturalmente la idea de un invariante p, que vale antes, durante y después del while, e incorpora una función t, conocida como *función cota*, que se define en términos de las variables de programa y cuyo dominio son los números naturales. La regla establece que la función se decrementa con cada iteración, lo que asegura que el while sea finito (el valor inicial de la función t representa la cantidad máxima de iteraciones).

He aquí el carácter no inductivo de esta regla. La función cota determina un *variante*, cuyo valor decreciente representa el acercamiento al evento de terminación, que se logra por la inexistencia de cadenas descendentes infinitas en el dominio de los números naturales con respecto a la relación <. El par (N, <) es un ejemplo de *orden bien fundado*: dominio con una relación anti-reflexiva, anti-simétrica y transitiva, sin cadenas que decrecen infinitamente. En efecto, propiedades como la terminación se prueban en el marco de un orden bien fundado.

La regla REP* tiene la siguiente forma:

$$\frac{\langle p \wedge B\rangle\ S\ \langle p\rangle\ ,\ \langle p \wedge B \wedge t = Z\rangle\ S\ \langle t < Z\rangle\ ,\ p \rightarrow t \geq 0}{\langle p\rangle \text{ while } B \text{ do } S \text{ od } \langle p \wedge \neg B\rangle}$$

La variable Z es una variable de especificación, no ocurre en p, B, t ni S, y su objetivo es conservar el valor de la expresión t antes de la ejecución del cuerpo del while. La primera premisa es la misma que la de la regla REP de H. Las otras dos premisas son las que aseguran la terminación:

- Por la segunda premisa, la expresión t se decrementa en cada iteración.
- Por la tercera premisa, t arranca y se mantiene no negativa después de cada iteración.

De esta manera el while debe terminar, y la postcondición es como antes $p \wedge \neg B$.

Con el método H* podemos demostrar directamente la correctitud total de un programa P con respecto a una especificación (p, q): si los invariantes de los while de P propagan la información necesaria para alcanzar al final de la prueba la postcondición q, no hace falta partir la prueba de $\langle p \rangle$ S $\langle q \rangle$ en las pruebas de {p} S {q} y $\langle p \rangle$ S $\langle true \rangle$ como indicamos antes. No obstante, trabajar separadamente suele ser más sencillo, porque los invariantes a considerar en las pruebas de terminación de los while suelen ser más simples. Con esta segunda alternativa en consideración, ejemplificamos a continuación el uso de la regla REP*.

Ejemplo de aplicación del método H*

Volvemos al programa de división entera:

$$S_{div} :: \ c := 0 \ ; r := x \ ; \text{while } r \geq y \text{ do } r := r - y \ ; c := c + 1 \text{ od}$$

para completar la verificación de su correctitud total. Ya probamos:

$$\{x \geq 0 \wedge y > 0\} \ S_{div} \ \{x = y \ . \ c + r \wedge r < y \wedge r \geq 0\}$$

Ahora probaremos:

$$\langle x \geq 0 \wedge y > 0 \rangle \ S_{div} \ \langle true \rangle$$

Alcanza con probar la terminación del único while del programa a partir de la precondición $x \geq 0 \wedge y > 0$. Como invariante proponemos $p = (y > 0 \wedge r \geq 0)$, más simple que el de la prueba de correctitud parcial, que era $x = y \ . \ c + r \wedge r \geq 0$. Y como función cota proponemos $t = r$. Notar que r se decrementa en cada iteración (en el valor de *y*), y que siempre se mantiene positivo.

Primero hay que probar que el invariante propuesto es precondición del while. En este caso:

a. $\langle x \geq 0 \wedge y > 0 \rangle$ c := 0 ; r := x $\langle p \rangle$

Y luego, para verificar la terminación del while, debemos probar las tres premisas establecidas por la regla REP*:

b. $\langle p \wedge r \geq y \rangle$ r := r – y ; c := c + 1 $\langle p \rangle$

c. $\langle p \wedge r \geq y \wedge r = Z \rangle$ r := r – y ; c := c + 1 $\langle r < Z \rangle$

d. $p \rightarrow r \geq 0$

Las pruebas de (a) a (d) no revisten mayor dificultad y quedan como ejercicio para el lector.

Sensatez del método H

Con el método H se prueban solamente fórmulas de correctitud parcial verdaderas, H es sensato, para todo programa S de PLW y todo par de aserciones p y q de Assn se cumple:

$$Tr \vdash_H \{p\}\ S\ \{q\} \rightarrow\ \models \{p\}\ S\ \{q\}$$

Como es habitual, para probar la sensatez de H es suficiente probar que sus axiomas son verdaderos y que sus reglas son sensatas, es decir que a partir de premisas verdaderas obtienen conclusiones verdaderas (tener en cuenta que el conjunto Tr de por sí contiene solo enunciados verdaderos). Empleamos inducción sobre la longitud de las pruebas, así que en la base de la inducción consideramos los axiomas, cuyas pruebas tienen un solo paso, y en el paso inductivo recurrimos a cada una de las reglas de H:

- Prueba de que el axioma SKIP es verdadero.
  Hay que probar $\models \{p\}$ skip $\{p\}$, es decir que a partir de un estado inicial que satisface p, luego de la ejecución del skip, si termina (un skip siempre termina), el estado final satisface la misma condición p.
  Se cumple porque por la semántica de PLW, (skip, σ) $\rightarrow$ (E, σ), y por lo tanto el estado inicial coincide con el estado final.
- La prueba de que el axioma ASI es verdadero es similar a la anterior y queda como ejercicio para el lector.
- Prueba de que la regla de la secuencia es sensata.
  Hay que probar $\models \{p\}\ S_1 ; S_2\ \{q\}$ si se prueban en H las fórmulas $\{p\}\ S_1\ \{r\}$ y $\{r\}\ S_2\ \{q\}$, o lo que es lo mismo, por la hipótesis inductiva, si se cumple $\models \{p\}\ S_1\ \{r\}$ y $\models \{r\}\ S_2\ \{q\}$.
  Sea σ_0 un estado inicial que satisface p y tal que $val(\pi(S_1 ; S_2, \sigma_0)) = \sigma_2 \neq \perp$. Veamos que el estado final σ_2 satisface q. Por la semántica de PLW, $(S_1 ; S_2, \sigma_0) \rightarrow^* (S_2, \sigma_1) \rightarrow^* (E, \sigma_2)$. Como $\models \{p\}\ S_1\ \{r\}$ por hipótesis, entonces σ_1 satisface r. Y como $\models \{r\}\ S_2\ \{q\}$ por hipótesis, entonces σ_2 satisface q.
- La pruebas de la sensatez del resto de las reglas son similares a la anterior y quedan como ejercicio para el lector.

Completitud del método H

La sensatez de un método de verificación es una propiedad de cumplimiento mandatorio, no tiene sentido contar con un método que genere fórmulas falsas. La propiedad recíproca, la completitud, es una propiedad deseable, se cumple si el alcance del método es total, si todas las fórmulas verdaderas son probables. En este caso significa que para todo programa S de PLW y todo par de aserciones p y q de Assn:

$$|= \{p\}\ S\ \{q\} \rightarrow Tr\ |\!-_H \{p\}\ S\ \{q\}$$

Ya indicamos que en términos absolutos el método H tal como fue definido no es completo. En las pruebas se necesitan los enunciados verdaderos de los números enteros. Por ejemplo, para probar {true} x := e {x = e}, si x no está en *e*, tenemos que contar con el enunciado true → e = e. Extender H con una axiomatización de los números enteros no alcanza, por su propia incompletitud. Por ejemplo, el conjunto de las fórmulas {true} skip {p} no es recursivamente numerable: si p no es demostrable tampoco lo es {true} skip {p}, porque por la regla SKIP habría que probar true → p, es decir p. La completitud del método H es relativa. Relativa al conjunto de los enunciados verdaderos de los enteros, y no solamente, sino también relativa a Assn y PLW, asumiendo como es de esperar que H se utilice con distintos lenguajes de especificación, lenguajes de programación e interpretaciones, porque como veremos enseguida, es necesario que puedan expresarse todas las aserciones intermedias de una prueba, o en otras palabras, que el lenguaje Assn sea *expresivo* con respecto a PLW y la interpretación de los números enteros, lo cual se demuestra. En la práctica, alcanza con que pueda expresarse, dados una precondición p y un programa S, la correspondiente postcondición *más fuerte*, es decir una aserción que denote el siguiente conjunto de estados:

$$post(p, S) = \{\sigma' \mid \exists\sigma: \sigma \models p \wedge val(\pi(S, \sigma)) = \sigma' \neq \perp\}$$

post(p, S) es el conjunto de los estados finales obtenidos por S a partir del conjunto de estados iniciales que satisfacen p. Lo de postcondición más fuerte se debe a que es la postcondición más precisa que se puede formular. Se puede considerar alternativamente la precondición *más débil*, dados un programa S y una postcondición q.

Con estas consideraciones, ahora sí probamos que H es completo. Para mostrar que se pueden probar todas las fórmulas {p} S {q}, es suficiente tratar las cinco formas que puede adoptar un programa S (skip, asignación, secuencia, selección condicional y repetición). Empleamos esta vez inducción sobre la estructura de S, así que en la base de la inducción consideramos las instrucciones atómicas y en el paso inductivo las instrucciones compuestas:

- Si |= {p} skip {q}, entonces Tr $|\!-_H$ {p} skip {q}.

Por la semántica de PLW se cumple (skip, σ) $\rightarrow$ (E, σ), y como |= {p} skip {q}, entonces si σ satisface p también satisface q, por lo que $p \rightarrow q$. Finalmente, los pasos {q} skip {q} y $p \rightarrow q$ constituyen una prueba de {p} skip {q} en H aplicando la regla CONS.

- La prueba considerando la asignación es similar a la anterior y queda como ejercicio para el lector.
- Si |= {p} S_1 ; S_2 {q}, entonces Tr $|\!-_H$ {p} S_1 ; S_2 {q}.
  Sea r la aserción que denota el conjunto post(p, S_1). Como |= {p} S_1 ; S_2 {q}, por la definición de post(p, S_1) y la semántica de PLW se cumple |= {p} S_1 {r} y |= {r} S_2 {q}. Por hipótesis inductiva las fórmulas {p} S_1 {r} y {r} S_2 {q} se prueban en H, las cuales constituyen una prueba de {p} S_1 ; S_2 {q} aplicando la regla SEC.
- Las pruebas considerando la selección condicional y la repetición son similares a la anterior y quedan como ejercicio para el lector. De todos modos amerita que hagamos algún comentario sobre la existencia de los invariantes de los while. A partir de la hipótesis |= {r} while B do S od {q}, hay que encontrar un invariante p que cumpla con $r \rightarrow p$, $(p \wedge \neg B) \rightarrow q$, y |= {$p \wedge B$} S {p}, porque por hipótesis inductiva y aplicando las reglas REP y CONS se llega a la prueba de {r} while B do S od {q} en H. El invariante p debe denotar el conjunto de todos los estados alcanzados por cualquier cantidad de iteraciones de S a partir de un estado inicial que satisfaga r. Una forma natural de p es, por lo tanto, la disyunción infinita $p_0 \vee p_1 \vee \ldots \vee p_k \vee \ldots$, con $p_0 = r$, y p_{i+1} denotando el conjunto post($p_i \wedge B$, S) para todo $i \geq 0$. Obviamente ésta no es una fórmula bien formada, pero se demuestra que p es expresable en Assn.

Sensatez del método H*

Solo hay que probar la sensatez de la regla REP*, porque los axiomas y las reglas restantes de H* son los mismos de H.

Supongamos que probamos en H* las premisas $\langle p \wedge B\rangle$ S $\langle p\rangle$, $\langle p \wedge B \wedge t = Z\rangle$ S $\langle t < Z\rangle$, y $p \rightarrow t \geq 0$. Por hipótesis inductiva se cumplen semánticamente.Y supongamos que a partir de la precondición p la repetición while B do S od no termina. Esto significa que a partir de un estado inicial σ_0 que satisface p, la computación π(while B do S od, σ_0) es infinita y tiene la forma:

$$(\text{while B do S od}, \sigma_0) \rightarrow^* (\text{while B do S od}, \sigma_1) \rightarrow^* \ldots \rightarrow^* (\text{while B do S od}, \sigma_i) \rightarrow^* \ldots$$

donde para todo $i \geq 0$, $\sigma_i \models p \wedge B \wedge t \geq 0$. Por lo tanto, la cadena $\sigma_0(t), \sigma_1(t), \ldots, \sigma_i(t), \ldots$ también es infinita, y como por hipótesis $\sigma_i(t) > \sigma_{i+1}(t)$ para todo $i \geq 0$, entonces obtuvimos una cadena descendente infinita en el orden bien fundado (N, <), lo que es absurdo. De esta manera el while termina, y lo hace en un estado σ_k que satisface $p \wedge \neg B$, tal como se establece en la prueba de sensatez del método H.

Completitud del método H*

Como en el caso de la sensatez, solo tenemos que considerar la repetición.

A partir de $\models \langle r \rangle$ while B do S od $\langle q \rangle$, hay que encontrar: (a) un invariante p que cumpla con $r \rightarrow p$, $(p \wedge \neg B) \rightarrow q$, y $\models \langle p \wedge B \rangle$ S $\langle p \rangle$, sobre el que ya hablamos en la prueba de la completitud de H, y (b) una función cota t que cumpla con $\models \langle p \wedge B \wedge t = Z \rangle$ S $\langle t < Z \rangle$, y $p \rightarrow t \geq 0$, porque por hipótesis inductiva y aplicando la regla CONS, se obtiene en H* la prueba de $\langle r \rangle$ while B do S od $\langle q \rangle$. Veamos qué necesidades de expresividad existen para (b).

Dada una repetición S :: while B do S_1 od, y una variable x no perteneciente a S, sea la siguiente repetición ampliada:

$$S_x :: x := 0 \text{ ; while B do } x := x + 1 \text{ ; } S_1 \text{ od}$$

Si S_x termina a partir de un estado σ en un estado σ', entonces σ'(x) es la cantidad de iteraciones de S a partir de σ. Haciendo iter(S, σ) = σ'(x), entonces claramente iter(S, σ) es una función parcial computable. Por lo tanto, la completitud de H* también se supedita a poder expresar iter(S, σ) con una expresión entera t tal que, para todo S y todo σ, si S termina a partir de σ entonces σ(t) = iter(S, σ), lo que se cumple si el sublenguaje de las expresiones enteras permite expresar la totalidad de las funciones parciales computables.

Programas secuenciales no determinísticos

En la sección anterior describimos dos lógicas de programas secuenciales determinísticos, con suficiente detalle para introducir adecuadamente los conceptos principales que pretendemos utilizar a lo largo de todo el capítulo. A partir de esta sección nos focalizamos en los aspectos distintivos de las lógicas para los paradigmas que faltan.

La programación no determinística facilita la abstracción, nos permite evitar detalles de implementación innecesarios en los programas, y así nos acerca más a lo que queremos expresar algorítmicamente. Semánticamente, ahora a partir de un estado inicial puede haber más de una computación, y por lo tanto más de un estado final. En particular, puede haber computaciones que terminan, fallan o no terminan. Los métodos de verificación de correctitud parcial y total deben contemplar estas diferencias con el paradigma anterior.

No se establece ninguna asunción de probabilidad de ejecución sobre las computaciones, salvo que se defina algún tipo de *fairness* (del inglés: justicia, ecuanimidad) para fijar restricciones sobre el comportamiento de las computaciones infinitas. Al final de esta sección describimos brevemente cómo impacta el fairness en la prueba de terminación.

Sintaxis del lenguaje de programación

Solo tenemos que describir el lenguaje de programación. Es una extensión no determinística de PLW. Se denomina GCL (por *guarded commands language* en inglés, es decir lenguaje de comandos con guardia o guardados). Este lenguaje lo utilizó E. Dijkstra para introducir la *derivación de programas* (construcción en simultáneo con la verificación, en

contraposición a la verificación *a posteriori*). Nosotros empleamos una variación sintáctica de dicho lenguaje. Las dos instrucciones que se agregan son la *selección no determinística*:

if B_1 then S_1 or B_2 then S_2 or … or B_n then S_n fi

y la repetición no determinística:

do B_1 then S_1 or B_2 then S_2 or … or B_n then S_n od

Para simplificar la escritura identificamos ambas instrucciones con IF y DO, respectivamente. La construcción B_i then S_i se conoce como *comando con guardia*. Los índices i se denominan *direcciones.* Si una guardia B_i es verdadera se dice que la dirección i está *habilitada*. Informalmente, la semántica del IF y del DO es:

- IF: Se evalúan sus guardias. Si algunas son verdaderas, se elige una no determinísticamente, se ejecuta la instrucción asociada, y se continúa con la instrucción siguiente al IF. Si en cambio todas las guardias son falsas, el IF (y el programa completo) termina en el estado de falla f.
- DO: Se evalúan sus guardias. Si algunas son verdaderas, se elige una no determinísticamente, se ejecuta la instrucción asociada, y se vuelve al comienzo del DO. Se continúa así hasta que ninguna guardia sea verdadera, en cuyo caso se continúa con la instrucción siguiente al DO. Por lo tanto en GCL el while y el DO son las fuentes de no terminación.

Por ejemplo, el siguiente programa GCL calcula, empleando el algoritmo de Euclides, el máximo común divisor de x > 0 e y > 0 (el resultado queda tanto en x como en y):

S_{mcd} :: do x > y then x := x – y or x < y then y := y – x od

Semántica del lenguaje de programación

Como antes, solo tenemos que referirnos al lenguaje de programación. Hay que definir la relación de transición $\rightarrow$ para las instrucciones IF y DO:

- Si $\sigma(B_i)$ = verdadero para algún i, entonces (IF, σ) $\rightarrow$ (S_i, σ)
  Si $\sigma(B_i)$ = falso para todo i, entonces (IF, σ) $\rightarrow$ (E, f)
- Si $\sigma(B_i)$ = verdadero para algún i, entonces (DO, σ) $\rightarrow$ (S_i ; DO, σ)
  Si $\sigma(B_i)$ = falso para todo i, entonces (DO, σ) $\rightarrow$ (E, σ)

Por el no determinismo del IF y del DO, ahora a partir de un estado σ y un programa S puede haber varias computaciones y estados finales. Π(S, σ) denota el conjunto de todas las

computaciones $\pi(S, \sigma)$ de un programa S a partir de un estado inicial σ, cada una de las cuales puede terminar, fallar o no terminar. Puede representarse como un *árbol de computaciones*, cuyas ramas son las computaciones $\pi(S, \sigma)$ y sus hojas los estados finales, que pueden ser estados propios, el estado de falla o el estado indefinido. La función semántica asociada a GCL es M: GCL $\rightarrow (\Sigma \rightarrow P(\Sigma))$, siendo $P(\Sigma)$ el conjunto de partes de Σ, y $M(S)(\sigma) = \{val(\pi(S, \sigma)) \mid \pi(S, \sigma) \in \Pi(S, \sigma)\}$.

Notar que el árbol de computaciones tiene grado finito, producto de la forma de las instrucciones IF y DO. Como consecuencia, se cumple la propiedad de *no determinismo acotado*: el conjunto de estados finales es finito o contiene al $\perp$. En efecto, la única posibilidad de que un programa GCL produzca un conjunto infinito de estados finales es que una de sus computaciones sea infinita, porque un árbol de grado finito con un número infinito de nodos tiene al menos una rama infinita (*lema de König*; se demuestra fácilmente por inducción, y el lector está invitado a probarlo).

Semántica de las fórmulas de correctitud

La posibilidad de que haya varias computaciones y estados finales, que pueden ser propios, el estado f o el estado $\perp$, hace que deba modificarse la semántica de las fórmulas {p} S {q} y $\langle p \rangle$ S $\langle q \rangle$ con respecto a lo definido para el paradigma determinístico. Ahora, un programa S es parcialmente correcto con respecto a una especificación (p, q) si y solo si para todo par de estados σ y σ':

$$(\sigma \models p \wedge \sigma' \in M(S, \sigma) \wedge \sigma' \neq f \wedge \sigma' \neq \perp) \rightarrow \sigma' \models q$$

Las computaciones que a partir de un estado que satisfaga la precondición p fallan o no terminan, no se consideran en la correctitud parcial, y las que terminan deben hacerlo en un estado que satisfaga la postcondición q. Por su parte, S es totalmente correcto con respecto a (p, q) si y solo si para todo par de estados σ y σ':

$$(\sigma \models p \wedge \sigma' \in M(S, \sigma)) \rightarrow (\sigma' \neq f \wedge \sigma' \neq \perp \wedge \sigma' \models q)$$

Todas las computaciones a partir de un estado que satisfaga p deben terminar, y deben hacerlo en un estado que satisfaga q. Sigue valiendo la posibilidad de expresar $\langle p \rangle$ S $\langle q \rangle$ con la conjunción {p} S {q} $\wedge$ $\langle p \rangle$ S $\langle true \rangle$ (la prueba queda como ejercicio para el lector).

Método D de verificación de correctitud parcial

La identificación de los métodos de esta sección con D y D* es en homenaje a Dijkstra. Solo tenemos que describir las reglas asociadas a las instrucciones IF y DO, las cuales reflejan las mismas ideas que las de las reglas del condicional y la repetición del paradigma determinístico:

- Regla del condicional (NCOND)

$$\frac{\{p \wedge B_i\}\ S_i\ \{q\},\ \ i = 1 \ldots n}{\{p\}\ IF\ \{q\}}$$

- Regla de la repetición (NREP)

$$\frac{\{p \wedge B_i\}\ S_i\ \{p\},\ \ i = 1 \ldots n}{\{p\}\ DO\ \{p \wedge \Lambda_{i=1,n} \neg B_i\}}$$

Ejemplo de aplicación del método D

Volviendo al programa S_{mcd} que calcula el máximo común divisor de x > 0 e y > 0, queremos probar:

{x = X ∧ y = Y ∧ X > 0 ∧ Y >0}
do x > y then x := x – y or x < y then y := y – x od
{x = mcd(X, Y)}

La interpretación de mcd(X, Y) es una función que devuelve el máximo común divisor de X e Y. Proponemos el invariante:

$$p = (mcd(x, y) = mcd(X, Y) \wedge x > 0 \wedge y > 0)$$

el cual refleja la idea del algoritmo de Euclides. La prueba se logra utilizando las reglas NREP y CONS, sin mayores dificultades (queda como ejercicio para el lector).

Método D* de verificación de correctitud total

En este caso, que no se cumpla la correctitud total puede deberse no solamente a que no termine una repetición determinística (while) o no determinística (DO), sino también a la falla de un IF, por tener todas sus guardias con el valor falso. Por lo tanto hay que modificar las reglas NCOND y NREP del método D. La regla NCOND* de D*, asociada a la instrucción IF, es:

$$\frac{p \longrightarrow V_{i=1,n}\ B_i\ ,\ \langle p \wedge B_i\rangle\ S_i\ \langle q\rangle,\ \ i = 1...n}{\langle p\rangle\ IF\ \langle q\rangle}$$

Es decir que la precondición del IF debe asegurar que al menos una guardia sea verdadera. La regla NREP* de D*, asociada a la instrucción DO, contiene las mismas premisas definidas en el caso determinístico, pero generalizadas a n guardias:

$$\frac{\langle p \wedge B_i\rangle\ S_i\ \langle p\rangle\ ,\ \langle p \wedge B_i \wedge t = Z\rangle\ S_i\ \langle t < Z\rangle\ ,\ i = 1...n,\ \ p \rightarrow t \geq 0}{\langle p\rangle\ DO\ \langle p \wedge \Lambda_{i=1,n} \neg B_i\rangle}$$

La variable Z no ocurre en p, t, B_i, S_i, para todo i, mientras que t es una expresión entera que denota una función natural definida en términos de las variables de programa.

Ejemplo de aplicación del método D*

Para probar $\langle x = X \wedge y = Y \wedge X > 0 \wedge Y > 0\rangle$ S_{mcd} $\langle x = mcd(X, Y)\rangle$, por el ejemplo previo alcanza con probar que S_{mcd} :: do x > y then x := x – y or x < y then y := y – x od, a partir de la precondición especificada, no falla y termina.

Que no falla es trivial porque el programa no tiene ningún IF.

Para la prueba de terminación del DO proponemos un invariante más simple que el utilizado para la prueba de correctitud parcial:

$$p = (x > 0 \wedge y > 0)$$

y la función cota:

$$t = x + y$$

que se decrementa iteración a iteración en x o en *y*, y siempre se mantiene por encima del cero. La prueba de las tres premisas de la regla NREP* se resuelve sin inconvenientes, y queda como ejercicio para el lector.

Sensatez y completitud de los métodos D y D*

Las pruebas de sensatez de las reglas NCOND, NCOND* y NREP son similares a las correspondientes del paradigma anterior. Se emplea inducción sobre la longitud de las pruebas. La salvedad es que ahora semánticamente se consideran computaciones no determinísticas.

Lo mismo sucede con la prueba de sensatez de la regla NREP*. Se llega a un absurdo asumiendo a pesar de las premisas que la repetición no termina, por tratarse con un orden bien fundado. En este caso la función cota t se decrementa iteración tras iteración con la ejecución de cualquier cuerpo S_i, el correspondiente a la guardia B_i seleccionada no determinísticamente.

Y también las pruebas requeridas en relación a dichas reglas para demostrar la completitud de D y D* siguen la misma línea de las que mostramos para H y H*, respectivamente. Se utiliza inducción estructural. La completitud es relativa al conjunto Tr de los enunciados verdaderos de los números enteros. El lenguaje Assn es expresivo con respecto a GCL y dicha interpretación. En particular se puede expresar el invariante p, que como antes, debe denotar el conjunto de todos los estados que se obtienen a lo largo de las iteraciones desde la precondición, esta vez con los cuerpos S_i pudiendo variar de iteración en iteración. Una fórmula infinita (no bien formada) que expresa p es entonces:

$$p = p_0 \vee p_1 \vee \ldots \vee p_k \vee \ldots$$

siendo $p_0 = r$, r la precondición, y para todo $i \geq 0$: p_{i+1} denotando la unión de los conjuntos post($p_i \wedge B_m$, S_m) para toda dirección m. En lo que hace a la función cota considerada en la regla NREP*, la expresión entera t es expresable en las mismas condiciones planteadas en REP*. Dada una instrucción DO que termina a partir de un estado σ, σ(t) debe entenderse como el número máximo de iteraciones. $\sigma(t) \in N$ porque el no determinismo es acotado; mostramos enseguida que con fairness esto deja de cumplirse.

Fairness

Incorporando fairness en la semántica, se restringe la manera en que pueden ocurrir las computaciones infinitas. Así, el fairness puede reducir el no determinismo. Está presente en varios lenguajes de programación concurrentes. Lo introducimos en esta sección porque su efecto puede ser mejor explicado en el contexto del no determinismo. Con la semántica operacional que venimos utilizando no hay una manera efectiva para definir todas y solo las computaciones de un programa permitidas por el fairness (computaciones *fair*), por lo que directamente se asume una restricción sobre su árbol de computaciones asociado: se *podan* sus ramas no fair. En la segunda parte de este capítulo describimos cómo con un lenguaje de la lógica temporal se pueden definir sintácticamente las computaciones fair.

Entre los tipos de fairness más usuales se encuentra el fairness *fuerte*, que no permite computaciones infinitas con una dirección infinitamente habilitada pero nunca seleccionada. Un caso particular de este tipo es el fairness *débil*, en que la dirección nunca seleccionada está continuamente habilitada a partir de un momento dado (las denominaciones fuerte y débil justamente se deben a que el primer tipo de fairness implica el segundo). Por ejemplo, el siguiente programa GCL no termina sin fairness pero sí lo hace con fairness débil (y por lo tanto también con fairness fuerte):

S_{nat} :: x := 0 ; b := true ; do 1: b then x := x +1 or 2: b then b := false od

Para facilitar el desarrollo, hemos ampliado el lenguaje GCL con variables booleanas (y expresiones booleanas que las contienen). Los identificadores de direcciones 1 y 2 no forman parte del lenguaje, los incluimos también para facilitar la explicación. Sin fairness, la computación infinita del programa S_{nat} que solo considera la dirección 1 está permitida, y por lo tanto el programa puede no terminar. En cambio con fairness débil, en algún momento debe seleccionarse la dirección 2, en cuyo caso la guardia b se hace falsa y el programa termina. De esta manera, S_{nat} con fairness débil produce como salida *cualquier* número natural. El siguiente programa GCL es una variación del anterior, que termina solo con fairness fuerte:

S_{par} :: x := 0 ; b := true ; do 1: b then x := x +1 or 2: b $\wedge$ par(x) then b := false od

El valor de la función par(x) es verdadero si y solo si el de la variable x es par. La dirección 2 está infinitamente habilitada pero de manera intermitente, por lo que el fairness débil no alcanza para que se seleccione alguna vez. Así, el programa S_{par} con fairness fuerte produce como salida *cualquier* número natural par. Notar que con el tipo de fairness indicado, los programas S_{nat} y S_{par} terminan y se comportan como generadores aleatorios de números naturales (el segundo solo de los pares). Esto significa que ahora, con fairness, tenemos un *no determinismo no acotado*, no se puede determinar a priori la cantidad máxima de iteraciones de un DO.

La correctitud total con fairness la expresamos con la fórmula $\langle\langle p\rangle\rangle$ S $\langle\langle q\rangle\rangle$, pudiendo agregar como superfijos en la pre y postcondición las letras f o d (por fuerte o débil) cuando por contexto no queda claro el tipo de fairness. Queda como ejercicio para el lector probar que también en este caso $\langle\langle p\rangle\rangle$ S $\langle\langle q\rangle\rangle$ equivale a la conjunción {p} S {q} $\wedge$ $\langle\langle p\rangle\rangle$ S $\langle\langle true\rangle\rangle$.

Un método usual para probar terminación con fairness es el de las *direcciones útiles*. La idea central es que ahora no es necesario que la función cota t se decremente cualquiera sea la dirección elegida, sino que es suficiente que lo haga solo atravesando algunas direcciones. En cada iteración se requiere que exista un conjunto de direcciones útiles, es decir que decrementen t, tales que el fairness fuerce a que se seleccionen alguna vez, al tiempo que las direcciones restantes no deben incrementar t. Por ejemplo, volviendo al programa S_{nat}:

S_{nat} :: x := 0 ; b := true ; do 1: b then x := x +1 or 2: b then b := false od

el cual termina con fairness débil, notar que la dirección útil es la 2. Una manera usual de expresar la función cota t en estos casos es en términos de la “distancia” a la terminación. Por la simplicidad de S_{nat} alcanza con que la distancia sea w = 1 al comienzo, cuando la guardia b es verdadera, y w = 0 cuando b se hace falsa. El fairness débil fuerza a que la dirección 2 se seleccione alguna vez. Correspondientemente, t podría valer 1 mientras se seleccione la dirección 1, y decrementarse a 0 cuando se tome la dirección 2 y así el programa termine. Una expresión adecuada para t podría ser entonces:

t = if b then 1 else 0 fi

Un ejemplo más ilustrativo, también con fairness débil, es el de un generador aleatorio de dos números naturales, x e y:

S_{2nat} :: x := 0 ; y : = 0 ; b_1 := true ; b_2 := true ;
do 1: $b_1 \wedge b_2$ then x := x +1 or
2: $b_1 \wedge b_2$ then b_1 := false or
3: $\neg b_1 \wedge b_2$ then y := y +1 or
4: $\neg b_1 \wedge b_2$ then b_2 := false od

En este caso se puede arrancar con una distancia w = 2, cuando las dos guardias son verdaderas, seguir con w = 1 cuando b_1 se hace falsa, y terminar con w = 0 cuando también se hace falsa b_2. Al comienzo la dirección útil es la 2, el fairness débil fuerza a que se seleccione alguna vez, y luego pasa a ser la 4, también obligadamente seleccionable por el fairness. Ahora, una expresión adecuada para t podría ser:

$$t = \text{if } b_1 \wedge b_2 \text{ then } 2 \text{ else } (\text{if } \neg b_1 \wedge b_2 \text{ then } 1 \text{ else } 0 \text{ fi}) \text{ fi}$$

Dado que con fairness el no determinismo es no acotado, no es suficiente que la función cota t varíe en el orden bien fundado (N, <). El número máximo de iteraciones se debe acotar con un número que sea mayor que todos los números naturales. Por eso se recurre al conjunto de los ordinales infinitos (W, <). De todos modos vemos que en los dos ejemplos previos alcanza con utilizar ({0, 1}, <) y ({0, 1, 2}, <), respectivamente.

Para probar terminación con fairness fuerte el método de las direcciones útiles es similar. En ambos casos, la sensatez se prueba del modo habitual: se asume que se prueban las premisas y que no hay terminación, y se llega al absurdo de encontrar una cadena descendente infinita en el orden bien fundado utilizado. Para la prueba de completitud, en cambio, se requiere el uso de otro lenguaje de aserciones.

Programas concurrentes

Los programas concurrentes ofrecen la posibilidad de computar datos más rápidamente que los programas secuenciales. Esta ventaja en la velocidad, que se torna sumamente necesaria en determinadas aplicaciones (administración de alarmas, soporte quirúrgico, pronóstico meteorológico, etc.), tiene como contrapartida que el desarrollo de los programas resulta en general dificultoso, los errores son más la regla que la excepción aún en componentes pequeños. De esta manera, la concurrencia más que ningún otro paradigma de programación requiere una metodología de prueba muy rigurosa. Debido a la habitual semántica de *intercalación* (*interleaving* en inglés) de las instrucciones atómicas de los distintos componentes secuenciales que se comunican y sincronizan, sin ninguna asunción de simultaneidad ni velocidad de ejecución, volvemos a encontrarnos con el no determinismo y así con varias computaciones y estados finales. Como novedades de esta sección tenemos, entre otras, problemas con la composicionalidad y la completitud, y más variedad de propiedades a probar, que también identificamos en la segunda parte de este capítulo.

Sintaxis del lenguaje de programación

Consideramos otra extensión de PLW, el lenguaje SVL (por *shared variables language* en inglés, es decir lenguaje de variables compartidas), que agrega la *composición concurrente*. Para simplificar, tratamos solo con programas de la forma:

$$S :: S_1 \| \ldots \| S_n$$

tal que || es el operador de concurrencia, y los S_i tienen instrucciones de PLW más eventualmente una instrucción de *retardo condicional* para la sincronización, la instrucción await. Se permite además un primer subprograma secuencial para inicialización de variables. La sintaxis del await es:

await B then S end

B es una expresión booleana y S tiene instrucciones de PLW a excepción del while (es decir que no hay anidamiento de instrucciones await y éstas siempre terminan). Los await se utilizan para sincronizar componentes, de modo tal que avancen solo al cumplirse determinados criterios de consistencia, o bien para obtener exclusividad sobre secciones críticas (secciones con variables compartidas modificables). La semántica informal de la instrucción await es la siguiente: (a) El subprograma asociado accede con exclusividad a las variables de B. (b) Si B es verdadera se ejecuta S atómicamente. (c) Si B es falsa se libera el acceso a las variables de B y el subprograma queda bloqueado hasta que más adelante en la misma situación pueda progresar.

Con respecto a la semántica informal del lenguaje SVL en general, destacamos:

- El control de un programa se sitúa en distintos lugares al mismo tiempo, uno en cada subprograma secuencial.
- El pasaje de información entre los distintos componentes de un programa se efectúa a través de las variables compartidas. La atomicidad definida para garantizar la consistencia de este pasaje se limita al skip, el await, y las asignaciones y evaluaciones de expresiones booleanas con una sola *referencia crítica*, es decir una sola variable compartida modificable. Así por ejemplo, la asignación x := x + 1 debe escribirse await true then x := x + 1 end, para que sea atómica. No obstante, para simplificar la notación evitaremos los await en las asignaciones y evaluaciones, entendiendo que se emplean cuando fuese necesario.
- Por la semántica de intercalación que se formaliza enseguida, un mismo programa puede producir computaciones que terminan, fallan (en este caso produciendo *deadlock* o bloqueo mortal, producto de un mal empleo de la instrucción await, que se manifiesta con uno o más subprogramas bloqueados indefinidamente), o no terminan. Todas las combinaciones de intercalación son posibles, a menos que se establezca algún tipo de fairness.

El siguiente es un ejemplo de programa SVL, que calcula en la variable n el factorial de un número natural N > 1. S_1 y S_2 identifican a los dos componentes que contribuyen al cálculo, uno haciendo 1 **.** 2 **.** 3 **.** … y el otro N **.** (N – 1) **.** (N – 2) **.** …, respectivamente:

S_{cfac} :: c_1 := true ; c_2 := true ; i := 1 ; k := N ; n := N ;

S_1 :: while c_1 do await true then

if i + 1 < k then i := i + 1 ; n := n . i else c_1 := false fi end od

||

S_2 :: while c_2 do await true then

if k – 1 > i then k := k – 1 ; n := n . k else c_2 := false fi end od

Semántica del lenguaje de programación

Completamos primero la semántica formal en el marco de los subprogramas secuenciales de SVL, con la semántica de la instrucción await. Se define:

$$\text{Si } \sigma \models B \text{ y } (S, \sigma) \rightarrow^* (E, \sigma'), \text{ entonces (await B then S end}, \sigma) \rightarrow (E, \sigma')$$

El await solo avanza si la expresión B es verdadera. Lo hace atómicamente, y siempre termina porque S no incluye instrucciones while.

Para definir la semántica formal en el marco de un programa SVL completo, recurrimos como siempre a la relación $\rightarrow$ para especificar las transiciones entre las configuraciones, ahora concurrentes, que son pares $(T_1||\ldots||T_n, \sigma)$, siendo T_i una continuación sintáctica del subprograma S_i, y σ el estado corriente, único para todos los componentes. La transición de una configuración concurrente a la siguiente se define de la siguiente manera:

$$\text{Si } (T_i, \sigma) \rightarrow (T'_i, \sigma'), \text{ entonces } (T_1||\ldots||T_i||\ldots||T_n, \sigma) \rightarrow_{(i,\sigma)} (T_1||\ldots||T'_i||\ldots||T_n, \sigma')$$

La expresión $\rightarrow_{(i,\sigma)}$ indica que la computación avanza por el componente i-ésimo, a partir del estado corriente σ. Más en general, se define:

$$(S_1||\ldots||S_n, \sigma) \rightarrow^*_h (T_1||\ldots||T_n, \sigma')$$

tal que $h = (i_1,\sigma)\ldots(i_k,\sigma_k)$, con $\sigma_k = \sigma'$, es la *historia* de las transiciones desde la configuración inicial $(S_1||\ldots||S_n, \sigma)$ hasta la configuración corriente $(T_1||\ldots||T_n, \sigma')$. Esta definición formaliza la semántica de intercalación. Se puede avanzar paso a paso por cualquier subprograma, a menos que se establezca algún tipo de fairness que restrinja la selección no determinística. Como antes, Π(S, σ) denota el conjunto de todas las computaciones π(S, σ) de un programa S a partir de un estado inicial σ, y también el árbol de computaciones asociado. Una computación termina si es finita y su configuración terminal tiene la forma $(E_1||\ldots||E_n, \sigma')$, falla (o tiene deadlock) si es finita y su configuración terminal no tiene dicha forma, o no termina si es infinita. La función semántica asociada a SVL es M: SVL $\rightarrow (\Sigma \rightarrow P(\Sigma))$, siendo $M(S)(\sigma) = \{val(\pi(S, \sigma)) \mid \pi(S, \sigma) \in \Pi(S, \sigma)\}$. M(S)(σ) puede tener estados propios, el estado de falla f y el estado indefinido $\perp$, y cumple que es finito o contiene al $\perp$, a menos que se asuma fairness.

Semántica de las fórmulas de correctitud

Se mantiene la semántica de las fórmulas {p} S {q} y ⟨p⟩ S ⟨q⟩ definida para el paradigma no determinístico, considerando ahora el lenguaje SVL en lugar de GCL, y que las fallas son por deadlock. Es decir, un programa S es parcialmente correcto con respecto a una especificación (p, q) si y solo si para todo par de estados σ y σ':

$$(\sigma \models p \wedge \sigma' \in M(S, \sigma) \wedge \sigma' \neq f \wedge \sigma' \neq \bot) \rightarrow \sigma' \models q$$

Por su parte, S es totalmente correcto con respecto a (p, q) si y solo si para todo par de estados σ y σ':

$$(\sigma \models p \wedge \sigma' \in M(S, \sigma)) \rightarrow (\sigma' \neq f \wedge \sigma' \neq \bot \wedge \sigma' \models q)$$

Queda como ejercicio para el lector probar que nuevamente podemos expresar la fórmula ⟨p⟩ S ⟨q⟩ con la conjunción {p} S {q} ∧ ⟨p⟩ S ⟨true⟩.

En el paradigma concurrente, a las propiedades básicas a verificar de correctitud parcial, ausencia de deadlock y terminación, se les suelen agregar otras dos, la *exclusión mutua* o *ausencia de interferencia* y la *ausencia de inanición* (*starvation* en inglés). En lo que sigue hacemos alguna mención a la prueba de estas propiedades adicionales.

Método O de verificación de correctitud parcial

El nombre del método proviene del de su creadora, S. Owicki. A los axiomas y reglas de H se les deben agregar en principio dos reglas (veremos que se necesita una más), una para el await y otra para la composición concurrente. La regla del await (AWAIT) es naturalmente similar a la del condicional (COND):

$$\frac{\{p \wedge B\}\ S\ \{q\}}{\{p\}\ \text{await } B \text{ then } S \text{ end}\ \{q\}}$$

El eventual deadlock producido por el await se considera en el método O* de verificación de correctitud total.

Para la composición concurrente, la forma natural de la regla asociada debería basarse en el mismo esquema composicional de la regla de la secuencia (SEC), es decir, a partir de las premisas {p1} S1 {q1}, … , {pn} Sn {qn}, debería derivarse la conclusión:

$$\{p_1 \wedge \ldots \wedge p_n\}\ S_1 \parallel \ldots \parallel S_n\ \{q_1 \wedge \ldots \wedge q_n\}$$

Pero lamentablemente la composicionalidad al menos así planteada se pierde en la concurrencia. A diferencia de la programación secuencial, no existe una conexión directa entre

las asignaciones que lleva a cabo un componente y los valores que obtiene cuando accede a las variables afectadas. Cuando las variables son compartidas, los valores también dependen de las asignaciones de otros componentes. Por ejemplo, se cumple:

$$\{x = 0\}\ S_1 :: x := x + 2\ \{x = 2\} \quad \text{y} \quad \{x = 0\}\ S_2 :: z := x\ \{z = 0\}$$

pero no se cumple:

$$\{x = 0\}\ S_1 :: x := x + 2 \ ||\ S_2 :: z := x\ \{x = 2 \wedge z = 0\}$$

porque al final también puede cumplirse $z = 2$. Es decir, la fórmula correcta es:

$$\{x = 0\}\ S_1 :: x := x + 2 \ ||\ S_2 :: z := x\ \{x = 2 \wedge (z = 0 \vee z = 2)\}$$

Más aún, esta última fórmula no se cumple si se reemplaza S_1 por el subprograma funcionalmente equivalente $S_3 :: x := x + 1 ; x := x + 1$. Efectivamente, para toda especificación (p, q) vale $\{p\}\ S_1\ \{q\} \leftrightarrow \{p\}\ S_3\ \{q\}$, a pesar de lo cual la fórmula:

$$\{x = 0\}\ S_3 :: x := x + 1 ; x := x +1 \ ||\ S_2 :: z := x\ \{x = 2 \wedge (z = 0 \vee z = 2)\}$$

no se cumple porque al final también puede darse la igualdad $z = 1$. Es decir, la fórmula correcta es en este caso:

$$\{x = 0\}\ S_3 :: x := x + 1 ; x := x +1 \ ||\ S_2 :: z := x\ \{x = 2 \wedge (z = 0 \vee z = 1 \vee z = 2)\}$$

Así, a diferencia de lo que sucede en la composición secuencial, en la composición concurrente dos subprogramas funcionalmente equivalentes no necesariamente son intercambiables, se pierde la noción de *caja negra*.

Existe una aproximación composicional, que requiere primero enriquecer la especificación de cada componente con información de la interacción con el resto, lo cual torna en general dificultoso construir la prueba. También hay aproximaciones que ignoran la estructura del programa, tratando directamente las computaciones como un todo (apoximación *global*), o bien transformando el programa concurrente original en un programa no determinístico equivalente (aproximación *reduccionista*). Con la aproximación global trabajamos en la segunda parte del capítulo. El problema con esta metodología es que al ignorar los componentes, no puede utilizarse como guía para la construcción sistemática de programas. Otra alternativa, muy difundida y que es la que presentamos en esta sección, sin ser composicional respeta la estructura modular del programa. La idea sigue siendo derivar a partir de $\{p_1\}\ S_1\ \{q_1\}, \ldots, \{p_n\}\ S_n\ \{q_n\}$, la fórmula $\{p_1 \wedge \ldots \wedge p_n\}\ S_1 \ ||\ \ldots\ ||\ S_n\ \{q_1 \wedge \ldots \wedge q_n\}$. Se plantea una prueba en dos etapas:

- En una primera etapa se construye una *proof outline* (esquema de prueba) para cada componente, que no es más que el componente anotado con una prueba del mismo utilizando el método H ampliado con la regla AWAIT. Se intercalan aserciones pre(S) y post(S) antes y después de cada instrucción S, respectivamente, entendiendo que las aserciones son verdaderas en esos puntos de control, considerándolos aisladamente.
- Y en la segunda etapa las proof outlines se consisten: se chequea que todas las aserciones sean verdaderas considerando ahora el programa completo, contemplando cualquier posible computación (se dice que las proof outlines deben ser *libres de interferencia*). Más precisamente: para toda aserción r de una proof outline y toda aserción pre(S) de otra, tal que S es un await o una asignación no incluida en un await (se la conoce como asignación *normal*), se debe cumplir la fórmula $\{r \wedge pre(S)\}\ S\ \{r\}$. En otras palabras, todas las aserciones de las proof outlines deben ser invariantes, cualquiera sea la computación concurrente ejecutada, y para ello basta con tener en cuenta solo las instrucciones que pueden modificar variables, que son el await y las asignaciones normales.

Justamente por la invariancia de las aserciones de las proof outlines, luego de ejecutarse el programa $S_1 \parallel \ldots \parallel S_n$ a partir de $p_1 \wedge \ldots \wedge p_n$ valdrá naturalmente $q_1 \wedge \ldots \wedge q_n$. No hay composicionalidad, no alcanza con utilizar fórmulas de correctitud, se debe analizar la estructura de los componentes y cómo interactúan entre ellos, los componentes deben tratarse como *cajas blancas*.

Las proof outlines {pre(S)} S {post(S)} se definen inductivamente de la siguiente manera:

- Si S :: skip, entonces $pre(S) \rightarrow post(S)$
- Si S :: x := e, entonces $pre(S) \rightarrow post(S)[x|e]$
- Si S :: S_1 ; S_2, entonces $pre(S) \rightarrow pre(S_1)$, $post(S_1) \rightarrow pre(S_2)$, $post(S_2) \rightarrow post(S)$
- Si S :: if B then S_1 else S_2 fi, entonces $pre(S) \wedge B \rightarrow pre(S_1)$, $pre(S) \wedge \neg B \rightarrow pre(S_2)$, $post(S_1) \rightarrow post(S)$, $post(S_2) \rightarrow post(S)$
- De modo similar se define para la repetición y el await (queda como ejercicio para el lector).

Así llegamos a la formulación de la regla de la composición concurrente (CONC):

$$\frac{\{p_1\}\ S_1\ \{q_1\}, \ldots, \{p_n\}\ S_n\ \{q_n\}, \text{ proof outlines libres de interferencia}}{\{p_1 \wedge \ldots \wedge p_n\}\ S_1 \parallel \ldots \parallel S_n\ \{q_1 \wedge \ldots \wedge q_n\}}$$

La regla CONC es en realidad una meta-regla, porque no tiene como premisas fórmulas de correctitud. El chequeo de que las proof outlines sean libres de interferencia tornan las pruebas muy trabajosas: dados dos subprogramas de tamaño n_1 y n_2, en la segunda etapa hay que

verificar $n_1 . n_2$ fórmulas de correctitud. De todas maneras en la práctica muchas validaciones se resuelven trivialmente. Un caso típico se da cuando la aserción r y la instrucción S no comparten variables. Otro caso es cuando resulta falsa la conjunción $r \wedge pre(S)$, situación que representa una computación que no puede suceder.

Ejemplo de aplicación del método O

Probamos a continuación $\{x = 0\}\ x := x + 1 \parallel x := x + 2\ \{x = 3\}$. Proponemos las siguientes proof outlines:

a. $\{x = 0 \vee x = 2\}\ x := x + 1\ \{x = 1 \vee x = 3\}$
b. $\{x = 0 \vee x = 1\}\ x := x + 2\ \{x = 2 \vee x = 3\}$

Una técnica habitual para obtener las proof outlines es debilitar las aserciones, considerando las distintas computaciones. Por ejemplo, en la precondición de la primera proof outline se reemplaza $x = 0$ por $x = 0 \vee x = 2$, teniendo en cuenta la asignación de la segunda proof outline. Hay que probar que las proof outlines son libres de interferencia, es decir que las siguientes fórmulas son verdaderas:

1. $\{(x = 0 \vee x = 2) \wedge (x = 0 \vee x = 1)\}\ x := x + 2\ \{x = 0 \vee x = 2\}$
2. $\{(x = 1 \vee x = 3) \wedge (x = 0 \vee x = 1)\}\ x := x + 2\ \{x = 1 \vee x = 3\}$
3. $\{(x = 0 \vee x = 1) \wedge (x = 0 \vee x = 2)\}\ x := x + 1\ \{x = 0 \vee x = 1\}$
4. $\{(x = 2 \vee x = 3) \wedge (x = 0 \vee x = 2)\}\ x := x + 1\ \{x = 2 \vee x = 3\}$

Queda como ejercicio para el lector comprobar que las proof outlines propuestas son correctas y libres de interferencia. Finalmente, como se cumple que la precondición del programa implica la conjunción de las precondiciones de las proof outlines, y la conjunción de las postcondiciones de las proof outlines implica la postcondición del programa, es decir:

5. $x = 0 \rightarrow ((x = 0 \vee x = 2) \wedge (x = 0 \vee x = 1))$
6. $((x = 1 \vee x = 3) \wedge (x = 2 \vee x = 3)) \rightarrow x = 3$

aplicando las reglas CONC y CONS se llega a $\{x = 0\}\ x := x + 1 \parallel x := x + 2\ \{x = 3\}$.

Extensión del método O

Ya anticipamos que el método O tiene una regla más. El debilitamiento de las aserciones, necesario para que las proof outlines sean libres de interferencia, atenta contra la completitud del método. En efecto, el método definido hasta ahora no es completo, y la nueva regla se introduce para que lo sea. Por ejemplo, claramente la fórmula $\{x = 0\}\ x := x + 1 \parallel x := x + 1\ \{x = 2\}$ es verdadera. Las proof outlines naturales para probarla son:

a. $\{x = 0 \vee x = 1\}\ x := x + 1\ \{x = 1 \vee x = 2\}$

b. $\{x = 0 \vee x = 1\}$ x := x + 1 $\{x = 1 \vee x = 2\}$

pero no son libres de interferencia ni conducen a la postcondición requerida (la comprobación de esto queda como ejercicio para el lector). En este caso las aserciones son demasiado débiles, con la única variable de programa x no alcanza para registrar la historia de la computación ejecutada, se avance por uno u otro componente el estado intermedio es el mismo, satisfaciendo x = 1 (cuando en el ejemplo anterior en un caso satisfacía x = 1 y en el otro x = 2, lo que explica por qué se logró la prueba). Cualquier intento con otro par de proof outlines es infructuoso, lo que se puede verificar formalmente. Suponiendo lo contrario obtenemos una contradicción:

1. $\{p_1\}$ x := x + 1 $\{q_1\}$	supuesta proof outline del primer componente
2. $\{p_2\}$ x := x + 1 $\{q_2\}$	supuesta proof outline del segundo componente
3. $x = 0 \rightarrow (p_1 \wedge p_2)$	para llegar por CONS a la precondición x = 0
4. $(q_1 \wedge q_2) \rightarrow x = 2$	para llegar por CONS a la postcondición x = 2
5. $\{p_1 \wedge p_2\}$ x := x + 1 $\{p_1\}$	por la libertad de interferencia
6. $\{p_1 \wedge p_2\}$ x := x + 1 $\{p_2\}$	por la libertad de interferencia
7. $(p_1 \wedge p_2) \rightarrow (q_1 \wedge q_2)[x \mid x+1]$	por 1 y 2
8. $(p_1 \wedge p_2) \rightarrow (p_1 \wedge p_2)[x \mid x+1]$	por 5 y 6
9. $(p_1 \wedge p_2) \rightarrow \forall x \geq 0 : p_1 \wedge p_2$	por 8
10. $(p_1 \wedge p_2) \rightarrow \forall x \geq 1 : q_1 \wedge q_2$	por 7 y 9
11. $x = 0 \rightarrow \forall x \geq 1 : x = 2$	por 3, 4 y 10, lo cual es falso

Intuitivamente podría observarse que el problema radica en el paso 8, la imposibilidad de registrar cuántas veces puede ser incrementada la variable x.

El objetivo de la nueva regla, que describimos enseguida, es poder ampliar el programa con *variables auxiliares* y nuevas asignaciones que permitan fortalecer las aserciones de las proof outlines, pero sin afectar el cómputo original. Para ilustrar la idea consideremos el ejemplo anterior. Ampliamos el programa con una variable *y* en el primer componente y una variable z en el segundo componente, las inicializamos en 0, y les asignamos el valor 1 una vez que el componente respectivo incrementa la variable x, de modo tal de contribuir a fortalecer la registración de la computación llevada a cabo. El fragmento inicial del programa ampliado puede anotarse del siguiente modo:

$$\{x = 0\} \ y := 0 \ ; \{x = 0 \wedge y = 0\} \ z := 0 \ ; \{x = 0 \wedge y = 0 \wedge z = 0\}$$

y las aserciones de las proof outlines se pueden reforzar así:

a'. $\{(x = 0 \wedge y = 0 \wedge z = 0) \vee (x = 1 \wedge y = 0 \wedge z = 1)\}$
await true then x := x + 1 ; y := 1 end

$\{(x = 1 \wedge y = 1 \wedge z = 0) \vee (x = 2 \wedge y = 1 \wedge z = 1)\}$

b'. $\{(x = 0 \wedge y = 0 \wedge z = 0) \vee (x = 1 \wedge y = 1 \wedge z = 0)\}$
await true then x := x + 1 ; z := 1 end
$\{(x = 1 \wedge y = 0 \wedge z = 1) \vee (x = 2 \wedge y = 1 \wedge z = 1)\}$

Es fácil comprobar que se llega a la prueba de {x = 0} S' {x = 2}, siendo S' el programa ampliado (queda como ejercicio para el lector). Y como las nuevas asignaciones no afectan al cómputo de x, entonces la prueba también vale para el programa original con la misma especificación.

La nueva regla del método O se denomina AUX (por las variables auxiliares) y tiene la siguiente forma:

$$\frac{\{p\}\ S\ \{q\}}{\{p\}\ S_{|A}\ \{q\}}$$

A es el conjunto de variables auxiliares, S el programa ampliado y $S_{|A}$ el programa original. Las variables de A solo forman parte de las nuevas asignaciones, y si aparecen en su parte derecha también aparecen en su parte izquierda. La postcondición q no incluye variables de A. No es necesaria esta misma restricción para la precondición p, porque las variables auxiliares pueden servir como variables de especificación.

Método O* de verificación de correctitud total

La correctitud total de un programa SVL implica su correctitud parcial, ausencia de deadlock y terminación. Al igual que para la prueba de correctitud parcial, el método O* plantea para la prueba del resto de las propiedades partir de proof outlines libres de interferencia (pueden ser distintas para cada propiedad). La prueba de ausencia de deadlock consiste en lo siguiente:

- Marcar en las proof outlines todos los casos posibles de deadlock. Esto se hace identificando tuplas $C_i = (\lambda_1, \ldots, \lambda_n)$, tantas como casos posibles de deadlock existan en un programa con n componentes. Las λ_k son etiquetas asociadas a un await o al final de un subprograma S_k (toda C_i debe tener al menos una etiqueta asociada a un await).
- Caracterizar semánticamente las tuplas C_i mediante aserciones δ_i, conocidas como *imágenes semánticas*, y probar que todas ellas son falsas. Las δ_i son conjunciones de aserciones δ_{ik} que cumplen: (a) Si la etiqueta λ_k de C_i está asociada a una instrucción T :: await B then U end, entonces $\delta_{ik} = pre(T) \wedge \neg B$, es decir, se niega B para plantear un hipotético caso de bloqueo. (b) Si la etiqueta λ_k de C_i está asociada al final de un subprograma S_k, entonces $\delta_{ik} = post(S_k)$. De este modo, si todas las imágenes semánticas resultan falsas significa que no existe ningún caso de deadlock.

Por ejemplo, sean las siguientes proof outlines de un esquema de programa SVL con tres componentes, en donde ya aparecen las etiquetas λ_k:

a. S_1 :: … {pre(T_1)} $\lambda_1 \rightarrow T_1$ :: await B_1 then U_1 end … $\lambda_2 \rightarrow$ {post(S_1)}
b. S_2 :: … $\lambda_3 \rightarrow$ {post(S_2)}
c. S_3 :: … {pre(T_2)} $\lambda_4 \rightarrow T_2$ :: await B_2 then U_2 end … $\lambda_5 \rightarrow${post(S_3)}

Los casos posibles de deadlock son $C_1 = <\lambda_1, \lambda_3, \lambda_4>$, $C_2 = <\lambda_1, \lambda_3, \lambda_5>$, y $C_3 = <\lambda_2, \lambda_3, \lambda_4>$, y las imágenes semánticas asociadas son $\delta_1 = (pre(T_1) \wedge \neg B_1) \wedge post(S_2) \wedge (pre(T_2) \wedge \neg B_2)$, $\delta_2 = (pre(T_1) \wedge \neg B_1) \wedge post(S_2) \wedge post(S_3)$, y $\delta_3 = post(S_1) \wedge post(S_2) \wedge (pre(T_2) \wedge \neg B_2)$. La prueba se completa verificando que las aserciones δ_1, δ_2 y δ_3 son falsas.

Con respecto a la prueba de terminación, a partir de proof outlines libres de interferencia el método establece dos cláusulas:

- Los valores de una función cota t utilizada para probar la terminación de un while de un componente no pueden incrementarse por efecto de una instrucción await o una asignación normal S de otro componente. Si esto ocurre significa que el while puede no terminar. Se plantea entonces chequear, para toda t de una proof outline y toda S de otra proof outline, que se cumpla $\langle t = Z \wedge pre(S)\rangle$ S $\langle t \leq Z\rangle$. Notar que un componente puede acortar la duración de un while.
- Todas las aserciones utilizadas en una proof outline para establecer el decrecimiento de una función cota deben ser libres de interferencia.

Ejemplos de aplicación del método O*

Se propone al lector probar ausencia de deadlock en el programa que calcula el factorial presentado al comienzo de la sección. La prueba es sencilla dado que las negaciones de las expresiones booleanas de los await resultan directamente falsas.

Aplicamos ahora el método O* para verificar terminación. La fórmula a probar es:

$$\langle x > 0 \wedge par(x)\rangle \text{ while } x > 2 \text{ do } x := x - 2 \text{ od } \| \; x := x - 1 \; \langle x = 1\rangle$$

y las proof outlines propuestas son:

a. $\langle x > 0\rangle$ while x > 2 do $\langle x > 2\rangle$ x := x – 2 $\langle x > 0\rangle$ od $\langle x = 1 \vee x = 2\rangle$
b. $\langle par(x)\rangle$ x := x – 1 $\langle impar(x)\rangle$

donde par(x) e impar(x) expresan que x es par o impar, respectivamente. El invariante del while es p = x > 0, y la función cota utilizada, no anotada en la proof outline, es t = x. Queda como ejercicio para el lector completar la prueba. Para remarcar la necesidad de la segunda cláusula establecida por el método O*, consideremos ahora este otro programa:

S_1 :: while x > 0 do y := 0 ; if y = 0 then x := x – 1 else y := 0 fi od
||
S_2 :: while x > 0 do y := 1 ; if y = 1 then x := x – 1 else y := 1 fi od

El programa puede no terminar. Es posible por ejemplo que se ejecute una computación infinita en que se alternen las asignaciones y := 0 e y := 1. Una función cota apropiada para las dos repeticiones es t = x, que aisladamente sirve para probar su terminación. Pero considerando las proof outlines en conjunto se observa que la segunda cláusula del método no se satisface. La aserción y = 0 necesaria antes del if en S_1 para garantizar el decrecimiento de x no se preserva debido a la asignación y := 1 de S_2. Lo mismo sucede en S_2 con la aserción y = 1 con respecto a la asignación y := 0 de S_1.

La prueba de terminación asumiendo fairness se basa en las mismas ideas desarrolladas en la sección anterior. Por ejemplo, a partir de x = 1 el programa:

while x = 1 do skip od || x := 0

termina con fairness débil (y también fuerte). Una función cota apropiada para el while es t = x, que en algún momento llega a 0 porque por el fairness la asignación x := 0 se ejecuta alguna vez.

Extensión del método O*

Otras dos propiedades que se suelen considerar en la verificación de un programa concurrente son la exclusión mutua, para asegurar que ningún subprograma manipule inadecuadamente variables que comparte con otro, y la ausencia de inanición, para asegurar que todo subprograma que compita con otro por un recurso lo obtenga alguna vez. La exclusión mutua, al igual que la correctitud parcial y la ausencia de deadlock, se clasifican como propiedades *safety* (del inglés: seguridad). La ausencia de inanición, como la terminación, pertenecen a la clase de las propiedades *liveness* (del inglés: vivacidad). Una caracterización informal muy conocida establece que las propiedades safety se asocian con “cosas malas que no pueden suceder” (resultado no deseado, bloqueo, interferencia, etc.), mientras que las propiedades liveness se asocian con “cosas buenas que van a suceder” (finitud, obtención de recurso, etc).

Las propiedades safety se distinguen porque se prueban empleando inducción. En particular, la prueba de exclusión mutua mediante el método O* se puede plantear de una manera muy similar a la de ausencia de deadlock (queda como ejercicio para el lector). La prueba de una propiedad liveness, en cambio, se basa en la utilización de un orden bien fundado. La ausencia de inanición particularmente se puede probar empleando el método O* de una manera muy similar a la prueba de terminación (también queda como ejercicio para el lector). Existen otras

caracterizaciones de estas dos clases de propiedades, una de ellas es sintáctica empleando un lenguaje de la lógica temporal, que se describe en la segunda parte de este capítulo.

Sensatez y completitud de los métodos O y O*

La sensatez de la regla AWAIT se prueba de la misma manera que probamos la sensatez de la regla COND del método H. La sensatez de la regla AUX se prueba trivialmente teniendo en cuenta que el programa ampliado tiene solo asignaciones a variables que no alteran el cómputo del programa original. CONC es una meta-regla, y por lo tanto para probar su sensatez se requiere una aproximación distinta de la que venimos utilizando. Las premisas no pertenecen al lenguaje de las fórmulas de correctitud, sino que se parte de un conjunto de proof outlines libres de interferencia. Se prueba inductivamente lo que se conoce como *preservación composicional*: toda instrucción S que se ejecuta a partir de un estado que satisface la precondición pre(S), termina en un estado que satisface la postcondición post(S). Esto se cumple por la libertad de interferencia. La base de la inducción considera la configuración inicial, en la que el estado inicial satisface la conjunción de las precondiciones pre(S_i). El paso inductivo contempla cualquier transición con la que se avanza por algún subprograma. Y se llega al estado final satisfaciendo la conjunción de las postcondiciones post(S_i).

La prueba de completitud del método O se basa en las mismas ideas desarrolladas en las secciones anteriores, salvo que ahora también contempla la expresividad de la historia de las computaciones empleando variables auxiliares.

También la sensatez y completitud del método O*, en lo que hace a la prueba de terminación, se prueban como en los paradigmas anteriores. La sensatez de la meta-regla para verificar ausencia de deadlock se deriva de la libertad de interferencia de las proof outlines y de la naturaleza de las aserciones que conforman las imágenes semánticas asociadas a las hipotéticas situaciones de deadlock. Cabe remarcar que encontrar una prueba de ausencia de deadlock depende fuertemente de cómo se definan las proof outlines, no necesariamente sirven las que establecen la correctitud parcial.

Lógica de programas reactivos

A diferencia de los programas de entrada/salida, el objetivo principal de los programas reactivos no es producir resultados sino interactuar perpetuamente con el entorno. Es el caso por ejemplo de los sistemas operativos y de programas creados para controlar procesos. Por lo tanto no corresponde especificarlos mediante un par de aserciones como hemos venido haciendo. Más aún, a partir de los trabajos de A. Pnueli, para verificar programas reactivos está ampliamente aceptado utilizar una lógica alternativa a la lógica clásica descripta en el primer y segundo capítulo, la lógica temporal (una de las instanciaciones posibles de la lógica modal introducida en el capítulo anterior), más adecuada para especificar y probar la variedad de propiedades que se plantean. Las fórmulas manipuladas por los métodos de verificación en

este contexto son directamente las de dicha lógica, las cuales deben ser satisfechas por las computaciones de los programas. En un sentido mantenemos así la idea de lógica de programas, porque las computaciones de los programas reactivos se asocian íntimamente con propiedades expresadas por fórmulas de la lógica temporal.

Manteniendo el criterio de uniformidad con lo desarrollado previamente, seguimos trabajando con el dominio semántico de los números enteros, y volvemos a considerar un modelo de ejecución concurrente con variables compartidas, esta vez con computaciones solo infinitas, y con un no determinismo provocado no solo implícitamente por la semántica de intercalación de las instrucciones atómicas, sino también explícitamente por instrucciones no determinísticas.

Repetimos básicamente la misma estructura de las secciones anteriores. Primero definimos los lenguajes de programación y especificación, y posteriormente presentamos un método de verificación. Ahora en particular tenemos que describir de cero los axiomas y reglas de la lógica temporal en la que el método se basa, pero contamos con lo estudiado en el capítulo precedente.

Lenguaje de programación

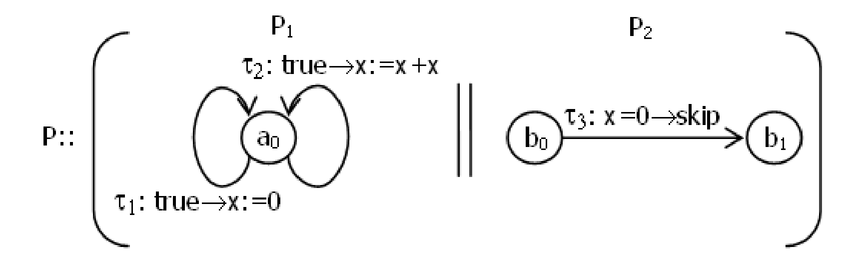
Consideramos una extensión de SVL conocida como SVT (por *shared variables text* en inglés, es decir texto de variables compartidas), que incluye entre otras cosas las instrucciones no determinísticas de GCL, una instrucción de agrupamiento ⟨S⟩ que permite ejecutar atómicamente una secuencia de instrucciones S, y la posibilidad de insertar etiquetas al comienzo y al final de las instrucciones. Por el enfoque de esta segunda parte, será más útil emplear directamente el modelo computacional asociado al lenguaje SVT, conocido como *sistema fair de transiciones*, que se suele representar con *diagramas de transiciones* (enseguida mostramos un ejemplo). El modelo consiste en un conjunto de *procesos* secuenciales que se ejecutan concurrentemente. La semántica es de intercalación de instrucciones atómicas, y la comunicación se efectúa a través de variables compartidas. Se distinguen los siguientes componentes:

- Un conjunto V de variables de programa x_1, $x_{2, \ldots,}$ y *variables de control* c_1, c_2, … Hay una variable de control por proceso, y sus valores son etiquetas que identifican locaciones en el mismo.
- Un conjunto Σ de estados σ_0, σ_1, …, que asignan valores a las variables de V.
- Un conjunto T de *transiciones* τ_1, τ_2, …, entre estados, cada una producto de la ejecución de una instrucción atómica. Para expresar la condición de progreso de una transición τ de σ a σ', es decir la condición para que τ esté *habilitada*, y la relación entre σ y su sucesor σ', se utiliza una aserción ρ(V, V'), o directamente ρ, conocida como *aserción de transición*, donde V y V' se refieren a los valores en σ y σ', respectivamente. Por ejemplo, la aserción

$\rho = (c_1 = a_0 \wedge x < 10 \wedge c'_1 = a_1 \wedge x' = x + 1 \wedge \forall y \in (V - \{x, c_1\}): y' = y)$ expresa que la transición correspondiente puede progresar desde la locación a_0 a la locación a_1 del proceso P_1 cuando el estado σ satisface la condición $x < 10$, quedando en el estado σ' el valor de x incrementado en 1 y los valores del resto de las variables sin modificar. Complementariamente, de manera similar a lo mostrado en la primera parte de este capítulo, a una transición τ se le puede asociar un par de aserciones (p, q): la terna {p} τ {q} es la *condición de verificación* de τ con respecto a p y q, y se cumple si vale la implicación $\rho \wedge p \longrightarrow q'$. Siguiendo con el ejemplo anterior, p podría ser $x = 0$ y q´ podría ser $x' = 1$.

- Una *condición inicial* Θ, que es una aserción que caracteriza los estados en los que pueden comenzar las computaciones (es la precondición).
- Una familia de *requerimientos de fairness débil* y una familia de *requerimientos de fairness fuerte*. Ambas familias son conjuntos de transiciones. Que una transición τ pertenezca a la primera o segunda familia establece la restricción de que no puede estar siempre o infinitas veces habilitada en una computación a partir de un momento dado sin ser ejecutada, respectivamente. La apertura en dos familias posibilita asignar requerimientos de fairness por tipo de transición (por ejemplo transición de sincronización, de comunicación, etc).

Una computación de un programa P de SVT es una secuencia infinita de estados $\pi = \sigma_0, \sigma_1, \ldots$ Denotamos como siempre con Π(P) al conjunto de computaciones de P. El siguiente es un ejemplo de diagrama de transiciones, que clarifica lo que hemos descripto:



El diagrama representa un programa P con dos procesos P_1 y P_2 que se ejecutan concurrentemente. Los nodos representan locaciones y los arcos transiciones. Sobre cada transición se explicita mediante un comando con guardia $B \longrightarrow S$ la instrucción a ser llevada a cabo, significando que si se cumple la condición B estando el control en la locación de origen, se puede ejecutar atómicamente la instrucción S y progresar a la locación de destino. Suponiendo que al comienzo $x = 10$, la condición inicial de P es:

$$\Theta = (x = 10 \wedge c_1 = a_0 \wedge c_2 = b_0)$$

P_1 puede progresar de a_0 a ella misma por la transición τ_1 o la transición τ_2 (la selección es no determinística), porque ambas transiciones están habilitadas. En un caso P_1 ejecuta $x := 0$, y

en el otro x := x + x. Por su parte, si x = 0, P_2 puede ejecutar el skip y progresar de b_0 a b_1. De esta manera, las aserciones ρ asociadas a τ_1, τ_2 y τ_3 son, respectivamente:

$$\rho_1 = (c_1 = a_0 \wedge \text{true} \wedge c'_1 = a_0 \wedge x' = 0 \wedge \forall y \in (V - \{x, c_1\})\text{: } y' = y)$$
$$\rho_2 = (c_1 = a_0 \wedge \text{true} \wedge c'_1 = a_0 \wedge x' = 2x \wedge \forall y \in (V - \{x, c_1\})\text{: } y' = y)$$
$$\rho_3 = (c_2 = b_0 \wedge x = 0 \wedge c'_2 = b_1 \wedge x' = x \wedge \forall y \in (V - \{x, c_2\})\text{: } y' = y)$$

Si se define por ejemplo que el conjunto completo de transiciones T está incluido en la familia de requerimientos de fairness débil, se cumple que el proceso P_2 termina.

Lenguaje de especificación

Presentamos un lenguaje de la lógica temporal para especificar programas SVT. El lenguaje permite expresar mediante *fórmulas temporales* propiedades que las computaciones de los programas (los *frames valuados* o *modelos*, usando terminología del capítulo anterior) deben satisfacer. Sin formalizar todavía su sintaxis y semántica, mostramos a continuación algunos ejemplos de fórmulas temporales, en todos los casos interpretadas sobre una computación π:

$$G\neg(SC_1 \wedge SC_2)$$

Si G es el operador temporal “siempre”, y SC_i significa que el proceso P_i está en su sección crítica, la fórmula especifica la propiedad de que P_1 y P_2 nunca ocupan sus secciones críticas en un mismo estado de π, es decir que expresa la exclusión mutua entre P_1 y P_2. Como segundo ejemplo sea la siguiente fórmula:

$$G(P_{1k} \rightarrow FU_{1k})$$

Si F es el operador temporal “alguna vez en el futuro”, y P_{1k} significa que el proceso P_1 solicita un recurso k y U_{1k} que P_1 utiliza k, la fórmula especifica que si la computación tiene un estado σ_i en el que P_1 solicita el recurso, entonces tiene también un estado σ_j, con $j \geq i$, en el que P_1 lo utiliza, es decir que expresa la ausencia de inanición de P_1 con respecto al recurso k. Los operadores G y F se pueden combinar para expresar el fairness débil y el fairness fuerte. Si H_i significa que la transición τ_i está habilitada y E_i que τ_i es ejecutada, entonces las fórmulas:

$$FGH_i \rightarrow GFE_i \quad \text{y} \quad GFH_i \rightarrow GFE_i$$

expresan que τ_i se ejecuta infinitas veces si a partir de un momento está habilitada continuamente o infinitas veces, respectivamente.

Un programa satisface una especificación si todas las fórmulas temporales que la componen son satisfechas por todas las computaciones del programa. Estructurar una especificación como una lista o conjunción de propiedades permite que pueda definirse incrementalmente, y así, si se la advierte incompleta, se la puede corregir agregándole las propiedades que le faltan.

El lenguaje de especificación se construye a partir del lenguaje de aserciones utilizado en las secciones anteriores, el lenguaje Assn (el dominio semántico sigue siendo el de los números enteros):

- Toda aserción del lenguaje Assn es una fórmula temporal. En este caso se conoce como *fórmula de estado*.
- El resto de las fórmulas temporales se generan a partir de otras utilizando operadores booleanos, operadores temporales y cuantificadores.

Describimos a continuación formalmente la sintaxis y semántica del lenguaje de especificación. En este último caso definimos, dada una computación π, cuándo una fórmula p se cumple en la posición $i \geq 0$ de π, lo que se denota con $(\pi, i) \models p$:

- Si p es una fórmula de estado, $(\pi, i) \models p \leftrightarrow \sigma_i \models p$. Naturalmente en este caso alcanza con evaluar p en el estado σ_i.
- $(\pi, i) \models \neg p \leftrightarrow (\pi, i) \not\models p$
- $(\pi, i) \models p \vee q \leftrightarrow (\pi, i) \models p \vee (\pi, i) \models q$. Las fórmulas $p \wedge q$, $p \longrightarrow q$ y $p \leftrightarrow q$ se pueden definir en términos de $\neg$ y $\vee$.

Los operadores temporales se particionan en dos grupos, *operadores de futuro* y *operadores de pasado*. Si bien la lógica temporal se puede definir sin pérdida de expresividad recurriendo únicamente a operadores de futuro, se consideran ambos grupos no solo para facilitar su uso, sino también porque por medio de los operadores de pasado se puede establecer una clasificación de propiedades caracterizada sintácticamente, que mostramos más adelante. Primero consideramos los operadores de futuro:

- $(\pi, i) \models Xp \leftrightarrow (\pi, i + 1) \models p$. X es el operador temporal de futuro *next* (siguiente). Xp se cumple en una posición si y solo si p se cumple en la posición siguiente.
- $(\pi, i) \models Gp \leftrightarrow (\pi, j) \models p$ para todo $j \geq i$. G es el operador temporal de futuro *globally* (globalmente o siempre). Gp se cumple en una posición si y solo si p se cumple en dicha posición y en todas las siguientes.
- $(\pi, i) \models Fp \leftrightarrow (\pi, j) \models p$ para algún $j \geq i$. F es el operador temporal de futuro *future* (futuro o alguna vez en el futuro). Fp se cumple en una posición si y solo si p se cumple en dicha posición o en alguna posición siguiente. F es dual del operador G: para todo p, Fp se cumple en una posición si y solo si en ella se cumple $\neg G \neg p$.

- $(\pi, i) \models pUq \leftrightarrow$ existe algún $j \geq i$ tal que $(\pi, j) \models q$, y para todo k, siendo $i \leq k < j$, se cumple $(\pi, k) \models p$. U es el operador temporal de futuro *until* (hasta). pUq se cumple en una posición i si y solo si: (a) q se cumple en dicha posición, o (b) q se cumple en alguna posición siguiente j y p se cumple en las posiciones i, i + 1, …, j – 1.
- $(\pi, i) \models pU_wq \leftrightarrow (\pi, i) \models pUq \vee (\pi, i) \models Gp$. U_w es el operador temporal de futuro *weak until* (hasta débil). Mientras que pUq garantiza que q ocurre en el futuro, pU_wq expresa una propiedad más débil: p se cumple hasta la siguiente ocurrencia de q o continuamente.

Los siguientes son ejemplos de fórmulas con operadores de futuro que suelen utilizarse:

- GFq. Esta fórmula establece que q se cumple infinitas veces.
- FGq. Esta fórmula establece que a futuro q se cumple permanentemente.
- $G(p \longrightarrow Gp)$. Esta fórmula establece que una vez que se cumple p, se cumple para siempre.
- $G(p \longrightarrow Fq)$. Esta fórmula establece que cuando se cumple p se cumple a futuro q.

Los operadores temporales de pasado constituyen la contraparte simétrica de los de futuro. Definimos:

- $(\pi, i) \models Yp \leftrightarrow i > 0 \wedge (\pi, i-1) \models p$. Y es el operador temporal de pasado *yesterday* (ayer o previo). Yp se cumple en una posición si y solo si la posición no es la primera y p se cumple en la posición anterior. Yp es falsa en la primera posición cualquiera sea p.
- $(\pi, i) \models Y_wp \leftrightarrow i = 0 \vee (i > 0 \wedge (\pi, i-1) \models p)$. Y_w es el operador temporal de pasado *yesterday weak* (ayer o previo débil), dual de la versión fuerte, porque $Y_wp = \neg Y \neg p$. Notar que para todo p, en la primera posición Y_wp es verdadera y ya vimos que Yp es falsa, mientras que en el resto de las posiciones se cumplen las dos fórmulas o ninguna.
- $(\pi, i) \models Hp \leftrightarrow (\pi, j) \models p$ para todo j tal que $0 \leq j \leq i$. H es el operador temporal de pasado *historically* (históricamente o siempre en el pasado). Hp se cumple en una posición si y solo si p se cumple en dicha posición y en todas las anteriores.
- $(\pi, i) \models Op \leftrightarrow (\pi, j) \models p$ para algún j tal que $0 \leq j \leq i$. O es el operador temporal de pasado *once* (una vez o alguna vez en el pasado). Op se cumple en una posición si y solo si p se cumple en dicha posición o en alguna anterior. O es dual del operador H: para todo p se cumple Op si y solo si se cumple $\neg H \neg p$.
- $(\pi, i) \models pSq \leftrightarrow$ existe algún j, con $0 \leq j \leq i$, tal que $(\pi, j) \models q$, y para todo k, siendo $j < k \leq i$, se cumple $(\pi, k) \models p$. S es el operador temporal de pasado *since* (desde). pSq se cumple en una posición i si y solo si: (a) q se cumple en dicha posición, o (b) q se cumple en alguna posición anterior j y p se cumple en las posiciones j + 1, j + 2, …, i.
- $(\pi, i) \models pS_wq \leftrightarrow (\pi, i) \models pSq \vee (\pi, i) \models Hp$. S_w es el operador temporal de pasado *weak since* (desde débil). pSq garantiza que q ocurre en el pasado, en cambio pS_wq establece que p se cumple desde la última ocurrencia de q o desde la primera posición.

Los siguientes son ejemplos de fórmulas con operadores de pasado que suelen utilizarse:

- ¬Y true. Esta fórmula establece que se está en la primera posición. A esta propiedad se la suele expresar con el predicado *first* (por primera posición). Por ser Y_w dual de Y, first también se puede expresar con Y_w false.
- G(p ⟶ Oq). Esta fórmula establece que siempre que se cumple p, en el pasado o en el presente también se cumple q.
- GO first. Esta fórmula establece que una posición es o está precedida por la primera.

Se define como fórmula de futuro a una fórmula que solo tiene operadores de futuro; expresa una propiedad sobre un sufijo σ_i, σ_{i+1}, … de una computación. Por su parte, una fórmula de pasado solo tiene operadores de pasado, que expresa una propiedad sobre un prefijo σ_0, …, σ_i de una computación. Las fórmulas de estado se consideran tanto de futuro como de pasado. Nos quedan por definir las fórmulas temporales con cuantificadores:

- $(\pi, i) \vDash \exists x{:}p \leftrightarrow (\pi', i) \vDash p$ para alguna computación π' tal que los estados de π y π' difieren a lo sumo en el valor de la variable x.
- $(\pi, i) \vDash \forall x{:}p \leftrightarrow (\pi', i) \vDash p$ para toda computación π' tal que los estados de π y π' difieren a lo sumo en el valor de la variable x.

Una fórmula temporal p es *válida* si para toda computación π se cumple $(\pi, 0) \vDash p$, o directamente $\pi \vDash p$. Se denota con $\vDash p$. En particular, considerando un determinado programa P, una fórmula temporal p es *P-válida* si se cumple $\pi \vDash p$ para toda computación π de Π(P). Se denota con $P \vDash p$. En lo que sigue presentamos una lógica temporal para probar, mediante una *parte general*, fórmulas válidas, y mediante una *parte de programa*, fórmulas P-válidas. La misma constituye un método de verificación de propiedades de programas SVT.

Método de verificación

Describimos primero la parte general del método, que provee axiomas y reglas para establecer la validez de fórmulas temporales, interpretadas en el dominio de los números enteros, sobre cualquier computación, sin asociación alguna con un programa determinado. Tanto en ésta como en la parte de programa, las fórmulas con símbolos p, q, r, …, deben entenderse como *esquemas de fórmulas*, los símbolos representan fórmulas arbitrarias. Por ejemplo:

$$G(Fp \leftrightarrow (p \vee XFp))$$

expresa el conjunto infinito de fórmulas obtenidas reemplazando p por cualquier fórmula temporal. Así, haciendo p = Fq, queda:

$$G(FFq \leftrightarrow (Fq \vee XFFq))$$

Para simplificar la notación, en lugar de $G(p \longrightarrow q)$ y $G(p \leftrightarrow q)$ usamos de ahora en más las abreviaciones $p \Rightarrow q$ y $p \Leftrightarrow q$, respectivamente.

Parte general del método

Adoptamos como operadores básicos al X, U_w, Y_w y S_w. En algunos casos, para simplificar la notación, utilizamos otros operadores que se pueden definir a partir de los operadores básicos de la siguiente manera (queda como ejercicio para el lector probar las igualdades):

$Gp = pU_w false$	$Hp = pS_w false$
$Fp = \neg G \neg p$	$Op = \neg H \neg p$
$pUq = pU_w q \wedge Fq$	$pSq = pS_w q \wedge Oq$
$Yp = \neg Y_w \neg p$	

Los siguientes axiomas se conocen como *axiomas de futuro*:

- AF0. $Gp \longrightarrow p$
- AF1. $X\neg p \Leftrightarrow \neg Xp$
- AF2. $X(p \longrightarrow q) \Leftrightarrow (Xp \longrightarrow Xq)$
- AF3. $G(p \longrightarrow q) \Rightarrow (Gp \longrightarrow Gq)$
- AF4. $Gp \longrightarrow GXp$
- AF5. $(p \Rightarrow Xp) \longrightarrow (p \Rightarrow Gp)$
- AF6. $(pU_w q) \Leftrightarrow (q \vee (p \wedge X(pU_w q)))$
- AF7. $Gp \Rightarrow pU_w q$

La validez de los axiomas precedentes no requiere mayores comentarios. Notar que en el axioma AF3 se establece la distribución del operador G con respecto a la implicación solo en un sentido, a diferencia del operador X en el axioma AF2. El axioma AF5 es una suerte de axioma de inducción. Los siguientes son los *axiomas de pasado* (en algún caso se utiliza el operador de futuro G):

- AP1. $Yp \Rightarrow Y_w p$
- AP2. $Y_w(p \longrightarrow q) \Leftrightarrow (Y_w p \longrightarrow Y_w q)$
- AP3. $H(p \longrightarrow q) \Rightarrow (Hp \longrightarrow Hq)$
- AP4. $Gp \longrightarrow GY_w p$
- AP5. $(p \Rightarrow Y_w p) \longrightarrow (p \Rightarrow Hp)$
- AP6. $(pS_w q) \Leftrightarrow (q \vee (p \wedge Y_w(pS_w q)))$
- AP7. $Y_w false$

También en este caso es sencillo comprobar la validez de los axiomas. No se incluyen las contrapartes de pasado de los axiomas AF0 y AF7, que son Hp ⟶ p y Hp ⇒ pS_wq, respectivamente, porque se pueden probar como teoremas. El axioma AP7 es el único que no se corresponde con un axioma de futuro. En los axiomas AP3 y AP5 aplican los mismos comentarios que hicimos de sus contrapartes de futuro. Existen además dos *axiomas mixtos*:

- AM0. p ⇒ XYp
- AM1. p ⇒ Y_wXp

También se cumple la recíproca del axioma AM0, es decir XYp ⇒ p, que se puede probar como teorema. El axioma AM1 es la contraparte de pasado del axioma AM0. El último axioma a presentar es el *axioma de las tautologías* (TAU), que permite incorporar como fórmulas válidas a todas las aserciones verdaderas del lenguaje Assn, es decir, todas las fórmulas de estado válidas (se cumplen en cualquier estado de cualquier computación). Así que nuevamente, tal como observamos en los métodos de verificación de la primera parte del capítulo, el presente método no es recursivo, salvo que nos restrinjamos a las tautologías proposicionales.

Las reglas de inferencia del método son las siguientes:

- Regla de generalización (GEN): Si p es una fórmula de estado válida, entonces Gp es una fórmula válida.
- Regla de especialización (ESP): Si Gp es una fórmula válida, siendo p una fórmula de estado, entonces p es una fórmula válida.
- Regla de instanciación (INST): Si p es una fórmula válida, entonces p[q|α] es una fórmula válida, tal que q es un símbolo de p, α es una fórmula, y p[q|α] denota la sustitución de q por α en la fórmula p. No confundir esta regla con la regla homónima presentada en una sección anterior.
- Regla de modus ponens (MP): Si p ⟶ q y p son fórmulas válidas, entonces q es una fórmula válida.

Queda como ejercicio para el lector probar la validez de los axiomas y la sensatez de las reglas. Los mismos corresponden al fragmento proposicional del lenguaje. Se prueba que esta parte del método es completa para probar la validez de cualquier fórmula temporal proposicional. No describimos la extensión que considera los elementos de primer orden (variables, igualdad, cuantificación), la cual, naturalmente, no preserva la completitud porque seguimos trabajando con el dominio de los números enteros.

Ejemplo de aplicación de la parte general del método

Probamos la fórmula $(Gr \wedge Fs) \Rightarrow (Fs \wedge Gr)$:

1. $(p \wedge q) \rightarrow (q \wedge p)$	(TAU)
2. $G((p \wedge q) \rightarrow (q \wedge p))$	(1, GEN)
3. $(p \wedge q) \Rightarrow (q \wedge p)$	(Definición de $\Rightarrow$)
4. $(Gr \wedge Fs) \Rightarrow (Fs \wedge Gr)$	(3, INST, $p \longleftarrow Gr$, $q \longleftarrow Fs$)

Parte de programa del método

Ya hemos indicado que expresar una especificación como una lista de propiedades facilita el chequeo de que sea completa. En este sentido resulta muy útil clasificar las propiedades, al menos las de uso más frecuente: dada una especificación, se revisa si mínimamente incluye propiedades de las clases más relevantes para el programa considerado. Así, una manera natural de presentar la parte de programa del método de verificación es describiendo las reglas de prueba de cada clase de propiedades.

Una clasificación referida previamente corresponde a la división entre propiedades safety y liveness. Se destaca: (a) las clases son disjuntas; (b) toda propiedad se puede expresar como la conjunción de una propiedad safety y una propiedad liveness; (c) las clases incluyen las propiedades que intuitivamente se asocian a ellas, como Gp en el caso de las propiedades safety, y Fq, $p \longrightarrow Fq$, GFq, etc., en el caso de las propiedades liveness; (d) las propiedades safety se prueban en base a un principio de invariancia, y las propiedades liveness recurriendo a un orden bien fundado; (e) toda computación en la que se viola una propiedad safety contiene un prefijo tal que todas sus extensiones infinitas también la violan, mientras que toda secuencia finita de estados $\sigma_0, \ldots, \sigma_k$ puede ser extendida a una computación infinita que satisface una propiedad liveness. Lamentablemente no existe una caracterización sintáctica para esta clasificación, existe para las propiedades safety pero solo para algunas subclases de propiedades liveness.

Para describir la parte de programa del método de verificación nos basamos en otra clasificación, que sí tiene una caracterización sintáctica. Se conoce como *clasificación safety-progreso*. A diferencia de la *clasificación safety-liveness*, sus clases no son disjuntas. Primero describimos suscintamente las clases de propiedades y luego las reglas del método:

- Las propiedades safety son las que se expresan con fórmulas de la forma Gp, siendo p una fórmula temporal de pasado. Ejemplos típicos de propiedades safety son la correctitud parcial, la exclusión mutua y la ausencia de deadlock. La clase es cerrada con respecto a la conjunción y la disyunción.
- Las propiedades *de garantía* son las que se expresan con fórmulas de la forma Fp, siendo p una fórmula temporal de pasado. Ejemplos típicos de propiedades de garantía son la terminación y la ocurrencia de un objetivo que se pretende alcanzar. La clase también es cerrada con respecto a la conjunción y la disyunción.

- Las propiedades de *intermitencia* u *obligación* son las que se expresan con fórmulas de la forma $Gp \vee Fq$, siendo p y q fórmulas temporales de pasado. Una forma equivalente muy utilizada es $Fp \longrightarrow Fq$. La clase es cerrada con respecto a la disyunción pero no lo es con respecto a la conjunción, e incluye de manera estricta a las clases de propiedades safety y de garantía. También se define la clase de propiedades de *intermitencia múltiple* u *obligación general*, con fórmulas de la forma $\Lambda_{i=1,k} (Gp_i \vee Fq_i)$, más amplia que la anterior.
- Las propiedades de *recurrencia* son las que se expresan con fórmulas de la forma GFp, siendo p una fórmula temporal de pasado. Una forma alternativa de uso frecuente es $G(p \longrightarrow Fq)$, equivalente a $GF(\neg pS_wq)$, y a la propiedad asociada se la identifica como propiedad de *respuesta*. Un ejemplo típico de propiedad de recurrencia es la ausencia de inanición. La clase es cerrada con respecto a la conjunción y la disyunción. Las fórmulas de recurrencia pueden especificar todas las propiedades safety, porque Gp es equivalente a GFHp, y todas las fórmulas de garantía, porque Fp es equivalente a GFOp. Tambien pueden expresar requerimientos de fairness débil. Si hab(τ) y eje(τ) son dos predicados que establecen que la transición τ está habilitada o es ejecutada, respectivamente, la fórmula correspondiente es: $G\ hab(\tau) \Rightarrow F\ eje(\tau)$, o equivalentemente: $GF(\neg hab(\tau) \vee eje(\tau))$. La clase incluye de manera estricta a la de las propiedades de intermitencia múltiple.
- Las propiedades de *persistencia* son las que se expresan con fórmulas de la forma FGp, siendo p una fórmula temporal de pasado. Usualmente estas fórmulas se utilizan para describir la eventual estabilización de un estado. Al igual que las fórmulas de recurrencia, las fórmulas de persistencia pueden especificar todas las propiedades safety porque Gp es equivalente a FGHp, y todas las fórmulas de garantía porque Fp es equivalente a FGOp. La clase es cerrada con respecto a la conjunción y la disyunción, y también incluye de manera estricta a la clase de las propiedades de intermitencia múltiple.
- Finalmente llegamos a la clase de las propiedades de *progreso*, que se expresan con fórmulas de la forma $GFp \vee FGq$, siendo p y q fórmulas temporales de pasado, y por lo tanto generalizan las propiedades de recurrencia y persistencia. Una forma alternativa de las fórmulas de progreso es $GFp \Rightarrow GFq$, por lo que son útiles para expresar requerimientos de fairness fuerte, como $GF\ hab(\tau) \Rightarrow GF\ eje(\tau)$. La clase es cerrada con respecto a la disyunción pero no con respecto a la conjunción. De este modo, las conjunciones de fórmulas de progreso, es decir $\Lambda_{i=1,n} (GFp_i \vee FGq_i)$, conocidas como fórmulas de *progreso múltiple*, constituyen la clase más amplia de la clasificación, que incluye a todas las demás. Más aún, se prueba que toda fórmula temporal sin cuantificadores es equivalente a una fórmula de esta clase (incluso se pueden considerar algunos casos especiales de fórmulas con cuantificadores).

No vamos a definir un conjunto de reglas de prueba para cada clase de propiedades, sino que nos concentraremos en tres casos en particular que reúnen las propiedades habitualmente más interesantes: las propiedades safety, de recurrencia y de progreso. Por lo visto recién, considerando reglas para probar las propiedades de progreso cubrimos el espectro completo

de la lógica temporal sin cuantificadores. Existen reglas (relativamente) completas para cada clase. A continuación describimos algunas de ellas. Se basan en la aproximación global, requieren programas completos para establecer sus propiedades. Existen reglas alternativas que permiten construir pruebas de una manera composicional. Comenzamos con una regla para probar propiedades safety de un programa:

Regla SAFE

$$\frac{\begin{array}{l} 1.\ (\theta \wedge \text{first}) \longrightarrow p \\ 2.\ p \Rightarrow q \\ 3.\ \{p\}\ T\ \{p\} \end{array}}{Gq}$$

En ésta y las siguientes reglas, T es el conjunto de transiciones, y {p} T {q} expresa que todas las transiciones τ de T cumplen la condición de verificación {p} τ {q}. Antes definimos esta condición con la implicación $(\rho \wedge p) \longrightarrow q'$, siendo p y q fórmulas de estado. Por tratar ahora con fórmulas temporales, usaremos $(\rho \wedge p) \Rightarrow q'$, ya que se requiere que la implicación se cumpla en todas las posiciones de una computación. La regla SAFE utiliza una fórmula auxiliar p, que por la premisa 1 se cumple inicialmente, y por la premisa 3 se propaga a lo largo de todo el programa. Por lo tanto p es un invariante. Como por la premisa 2 siempre p implica q, entonces q también es un invariante del programa. De las reglas para probar propiedades de recurrencia presentamos tres, sin y con asunción de fairness:

Regla REC

$$\frac{\begin{array}{l} 1.\ p \Rightarrow (q \vee r) \\ 2.\ \{r\}\ T\ \{q\} \\ 3.\ r \Rightarrow hab(T) \end{array}}{p \Rightarrow Fq}$$

El predicado hab(T) establece que todas las transiciones de T están habilitadas. Claramente si se cumplen las premisas, luego de a lo sumo una transición se obtiene la conclusión. La segunda regla contempla fairness débil:

Regla W-REC

$$\frac{\begin{array}{l} 1.\ p \Rightarrow (q \vee r) \\ 2.\ \{r\}\ T_1\ \{q\} \\ 3.\ \{r\}\ T_2\ \{q \vee r\} \\ 4.\ r \Rightarrow hab(T_1) \end{array}}{p \Rightarrow Fq}$$

T_1 y T_2 constituyen una partición de T, y T_1 es una familia de requerimientos de fairness débil. Si a partir de que se cumple p nunca se cumple q, entonces r se cumple continuamente y

no se ejecuta ninguna transición de T_1. Pero esto no puede ocurrir por la premisa 4 y la asunción de fairness débil. La tercera regla difiere de la anterior solo en su premisa 4, ahora contempla fairness fuerte:

Regla S-REC

1. $p \Rightarrow (q \vee r)$
2. $\{r\}\ T_1\ \{q\}$
3. $\{r\}\ T_2\ \{q \vee r\}$
4. $r \Rightarrow F(q \vee hab(T_1))$

$p \Rightarrow Fq$

T_1 y T_2 constituyen una partición de T, y T_1 es una familia de requerimientos de fairness fuerte. En este caso no puede suceder que a partir de p nunca se cumpla q, por la premisa 4 y la asunción de fairness fuerte. Mostramos por último una regla para probar propiedades de progreso. De las formas alternativas para expresar estas propiedades consideramos la fórmula $GFr \Rightarrow Fq$, pero extendida de la siguiente manera: $(p \wedge GFr) \Rightarrow Fq$. Esta fórmula establece que toda ocurrencia de p, seguida por infinitas ocurrencias de r, debe ser seguida por q. La regla contempla fairness fuerte:

Regla S-PROG

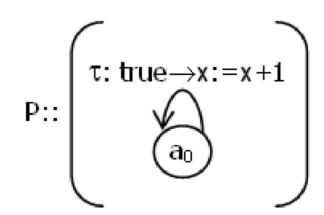
1. $p \Rightarrow (q \vee s)$
2. $\{s\}\ T_1\ \{q\}$
3. $\{s\}\ T_2\ \{q \vee s\}$
4. $(s \wedge GF(s \wedge r)) \Rightarrow F(q \vee hab(T_1))$

$p \wedge GFr \Rightarrow Fq$

T_1 y T_2 constituyen una partición de T, y T_1 es una familia de requerimientos de fairness fuerte. La justificación de la sensatez de esta regla es similar a la de la regla anterior (se invita al lector a probarla, al igual que la sensatez de las reglas anteriores de esta parte del método).

Ejemplo de aplicación de la parte de programa del método

Vamos a verificar una propiedad safety en un programa muy sencillo, representado por el siguiente diagrama de transiciones:



El programa P tiene un solo proceso, que transición tras transición incrementa el valor de la variable x en 1. La condición inicial es $\Theta = (x = 0 \wedge c = a_0)$. Probaremos que en la única computación de P se cumple:

$$G(x = 10 \rightarrow O(x = 5))$$

Recurrimos a la regla SAFE, así que debemos probar las premisas: (1) $(\Theta \wedge first) \longrightarrow p$, (2) $p \Rightarrow q$, y (3) {p} T {p}, para llegar a la conclusión Gq, siendo entonces q en este caso: $x = 10 \rightarrow O(x = 5)$. Como invariante p proponemos: $x \geq 5 \rightarrow O(x = 5)$. De esta manera, tenemos que demostrar:

1. $(x = 0 \wedge c = a_0 \wedge first) \longrightarrow (x \geq 5 \rightarrow O(x = 5))$
2. $(x \geq 5 \rightarrow O(x = 5)) \Rightarrow (x = 10 \rightarrow O(x = 5))$
3. $\{x \geq 5 \rightarrow O(x = 5)\}\ x := x + 1\ \{x \geq 5 \rightarrow O(x = 5)\}$

Prueba de (1):

Por el axioma TAU se obtiene $(x = 0 \wedge r \wedge s) \rightarrow (x \geq 5 \rightarrow t)$. La prueba se completa aplicando la regla INST con las sustituciones correspondientes.

Prueba de (2):

Por el axioma TAU se obtiene $(x \geq 5 \rightarrow r) \longrightarrow (x = 10 \rightarrow r)$. La prueba se completa aplicando primero la regla GEN, y luego la regla INST con las sustituciones correspondientes.

Prueba de (3):

La condición de verificación es la siguiente:

$$(x' = x + 1 \wedge (x \geq 5 \rightarrow O(x = 5))) \Rightarrow (x \geq 5 \rightarrow O(x = 5))'$$

Podemos reemplazar $(x \geq 5 \rightarrow O(x = 5))'$ por $(x' \geq 5 \rightarrow (x' = 5 \vee O(x = 5))$. Así, por el axioma TAU se obtiene $(x' = x + 1 \wedge (x \geq 5 \rightarrow r)) \rightarrow (x' \geq 5 \rightarrow (x' = 5 \vee r))$, y la prueba se completa aplicando la regla GEN y la regla INST con las sustituciones correspondientes.

Ejercicios

Programas secuenciales determinísticos

1. Completar la definición de las funciones semánticas V, W y T, de modo tal que sean funciones totales. *Ayuda: deben considerar el estado indefinido $\perp$ y el estado de falla f.*

2. La semántica de las expresiones enteras y booleanas del lenguaje PLW se definió denotacionalmente (funciones semánticas V y W). Plantear una semántica operacional para las mismas.
3. Desarrollar las distintas formas que pueden adoptar las computaciones de los programas PLW, a partir de la semántica operacional definida.
4. Probar que la correctitud total de un programa S de PLW con respecto a una especificación (p, q), es decir ⟨p⟩ S ⟨q⟩, se puede expresar con la conjunción {p} S {q} ∧ ⟨p⟩ S ⟨true⟩.
5. Probar $\{x \geq 0 \wedge y \geq 0\}\ S_{div}\ \{x = y \,.\, c + r \wedge r < y \wedge r \geq 0\}$, es decir la correctitud parcial en términos del programa de división entera y la especificación planteados antes, salvo que ahora el divisor *y* puede ser 0.
6. Completar la prueba de terminación del programa de división entera S_{div}.
7. Completar la prueba de sensatez del método H.
8. Probar que el método H sigue siendo sensato cuando se le incluyen el axioma INV y las reglas AND, OR e INST.
9. Completar la prueba de completitud del método H.

Programas secuenciales no determinísticos

10. Completar la sintaxis y la semántica del lenguaje GCL agregándole variables booleanas.
11. Probar el lema de König.
12. Probar que la correctitud total de un programa S de GCL con respecto a una especificación (p, q), es decir ⟨p⟩ S ⟨q⟩, se puede expresar con la conjunción {p} S {q} ∧ ⟨p⟩ S ⟨true⟩.
13. Probar la correctitud parcial del programa del máximo común divisor S_{mcd} con respecto a la especificación (x = X ∧ y = Y ∧ X > 0 ∧ Y >0, x = mcd(X, Y)).
14. Probar la correctitud total del programa S_{mcd} con respecto a la misma especificación planteada en el ejercicio anterior.
15. Desarrollar las pruebas de sensatez y completitud de los métodos D y D*.
16. Probar que si existe fairness fuerte entonces también existe fairness débil.
17. Probar que asumiendo fairness sigue valiendo que ⟨p⟩ S ⟨q⟩ se puede expresar con la conjunción {p} S {q} ∧ ⟨p⟩ S ⟨true⟩.
18. Formalizar el método D* asumiendo fairness débil a partir de los conceptos planteados previamente, y probar su sensatez.

Programas concurrentes

19. Probar que la correctitud total de un programa S de SVL con respecto a una especificación (p, q), es decir ⟨p⟩ S ⟨q⟩, se puede expresar con la conjunción {p} S {q} ∧ ⟨p⟩ S ⟨true⟩.

20. Completar la definición inductiva de una proof outline.

21. Completar la prueba de {x = 0} x := x + 1 || x := x + 2 {x = 3}.

22. Comprobar la incorrección de las proof outlines propuestas para la prueba de {x = 0} x := x + 1 || x := x + 1 {x = 2}.

23. Completar la prueba de {x = 0} x := x + 1 || x := x + 1 {x = 2} utilizando variables auxiliares.

24. Completar la prueba de $\langle x > 0 \wedge par(x)\rangle$ while x > 2 do x := x – 2 od || x := x – 1 $\langle x = 1\rangle$.

25. Probar la correctitud parcial, la ausencia de dealock y la terminación del programa S_{cfac} de SVL que calcula el factorial de N > 1. *Ayuda: podría considerarse como invariante de S_1 la aserción $p_1 = (i \leq j) \wedge (\neg c_1 \longrightarrow i + 1 = j) \wedge (n \,.\, (i+1) \,.\, \ldots \,.\, (j - 1) = N!)$, y de S_2 la aserción p_2 que es la misma que p_1 pero con c_2 en lugar de c_1.*

26. Extender el método O* para que también permita probar exclusión mutua y ausencia de inanición. *Ayuda: basarse, respectivamente, en las pruebas de ausencia de deadlock y terminación*.

27. Desarrollar las pruebas de sensatez de los métodos O y O*.

Programas reactivos

28. Probar las igualdades: (a) $Gp = pU_w false$, (b) $Hp = pS_w false$, (c) $Fp = \neg G \neg p$, (d) $Op = \neg H \neg p$, (e) $pUq = pU_w q \wedge Fq$, (f) $pSq = pS_w q \wedge Oq$, y (g) $Yp = \neg Y_w \neg p$.

29. Probar la sensatez de la parte general del método de verificación presentado.

30. Probar la fórmula $(Gr \wedge Fs) \longrightarrow Fs$.

31. Probar que de las premisas $p \Rightarrow q$ y $q \Rightarrow r$, se deduce la conclusion $p \Rightarrow r$.

32. Probar la sensatez de las reglas descriptas en la parte de programa del método de verificación presentado.

33. Probar que la clase de las propiedades safety es cerrada con respecto a la conjunción y la disyunción. *Ayuda: probar que $Gp \wedge Gq$ es equivalente a $G(p \wedge q)$, y que $Gp \vee Gq$ es equivalente a $G(Hp \vee Hq)$*.

34. Probar que la clase de las propiedades de garantía es cerrada con respecto a la conjunción y la disyunción. *Ayuda: probar que $Fp \wedge Fq$ es equivalente a $F(Op \wedge Oq)$, y que $Fp \vee Fq$ es equivalente a $F(p \vee q)$*.

35. Probar que la clase de las propiedades de intermitencia u obligación es cerrada con respecto a la disyunción. *Ayuda: probar que* $(Gp_1 \vee Fq_1) \vee (Gp_2 \vee Fq_2)$ *es equivalente a* $(Gp_1 \vee Gp_2) \vee (Fq_1 \vee Fq_2)$.

36. Probar que la clase de las propiedades de recurrencia es cerrada con respecto a la conjunción y la disyunción. *Ayuda: probar que* $GFp \wedge GFq$ *es equivalente a* $GF(q \wedge Y(\neg q\ S\ p))$*, y que* $GFp \vee GFq$ *es equivalente a* $GF(p \vee q)$.

37. Probar que la clase de las propiedades de persistencia es cerrada con respecto a la conjunción y la disyunción. *Ayuda: probar que* $FGp \wedge FGq$ *es equivalente a* $FG(p \wedge q)$*, y que* $FGp \vee FGq$ *es equivalente a* $FG(q \vee Y(p\ S\ (p \wedge \neg q)))$.

38. Probar que la clase de las propiedades de progreso es cerrada con respecto a la disyunción. *Ayuda: probar que* $(GFp_1 \vee FGq_1) \vee (GFp_2 \vee FGq_2)$ *es equivalente a* $(GFp_1 \vee GFp_2) \vee (FGq_1 \vee FGq_2)$.

Referencias y notas

Además de las notas en forma de apartados, con sus respectivas referencias bibliográficas, que exhibimos a continuación, remitimos al lector a la lectura de las notas, referencias bibliográficas y numerosos ejercicios de (Rosenfeld & Irazábal, 2013) y (Rosenfeld & Irazábal, 2010) que se refieren a la verificación de programas, material que constituye la bibliografía de cabecera de las materias Teoría de la Computación y Verificación de Programas y Teoría de la Computación y Verificación de Programas Avanzada, respectivamente. Ambos libros a su vez están basados esencialmente y de manera parcial en los contenidos de (Francez, 1992), (Apt, 1981), (Apt & Olderog, 1997), (de Bakker, 1980), (Manna & Pnueli, 1989), (Manna & Pnueli, 1992), (Manna & Pnueli, 1995) y (Hute & Ryan, 2004). Recomendamos también su lectura.

Particularmente en (Rosenfeld & Irazábal, 2013) se tratan los métodos de verificación de los programas secuenciales, las pruebas de su sensatez y completitud, distintos tipos de sensatez y completitud (en lo que sigue dedicamos un apartado sobre esto), la verificación de programas utilizando arreglos y procedimientos, y el desarrollo sistemático de programas tomando como base los métodos axiomáticos.

Por su parte en (Rosenfeld & Irazábal, 2010) se estudian los métodos de verificación de los programas no determinísticos con los paradigmas *control driven* y *data driven*, la verificación de los programas concurrentes con los modelos de memoria compartida y pasaje de mensajes, la verificación de programas con fairness fuerte y débil, una introducción a la semántica denotacional considerando el lenguaje PLW, y la verificación de programas reactivos usando lógica temporal, tanto líneal como ramificada, incluyendo una breve referencia a la verificación automática cuando la lógica es proposicional (hay también a continuación un apartado acerca de estos últimos tópicos).

Incompletitud

A lo largo del capítulo se ha establecido que la completitud de los distintos métodos de verificación estudiados, considerando la interpretación de los números enteros, no puede ser *absoluta* debido a la inexistencia de un método axiomático para demostrar todos los enunciados verdaderos de los enteros, lo que fue probado por K. Gödel en su famoso *teorema de incompletitud*, mencionado previamente. (La formulación general de) este teorema establece que todo método axiomático recursivo y consistente, con suficiente aritmética, tiene algún enunciado indecidible. Un método *recursivo* tiene un conjunto recursivo de axiomas, la única manera para que una demostración pueda corroborarse o refutarse mecánicamente (en los métodos de verificación de programas estudiados, justamente completos *relativamente*, se incorporan como axiomas todas los enunciados verdaderos de los enteros, que constituyen un conjunto ni siquiera recursivamente numerable). En un método *consistente* no se puede probar un enunciado φ y su negación ¬φ (si no sería un método inútil, permitiría probar cualquier enunciado). *Suficiente aritmética* alude a que a partir de los axiomas pueden demostrarse todos los enunciados de la aritmética *finitista*, es decir los enunciados cuya verdad o falsedad puede determinarse en una cantidad finita de pasos. Finalmente, un método tiene algún enunciado *indecidible* φ si en él no puede demostrarse ni φ ni su negación ¬φ (por lo tanto contiene un enunciado verdadero no demostrable en el método).

Gödel probaba así, en 1931, la imposibilidad de llevar a cabo el *programa* de D. Hilbert, con el que se pretendía formalizar completamente la matemática clásica reemplazando sus conceptos por cadenas de símbolos y el razonamiento por mera manipulación de dichas cadenas considerando reglas mecánicas. El teorema se formuló inicialmente con el requerimiento de *ω-consistencia*, condición más fuerte que la consistencia, pero posteriormente, en base a un desarrollo más complejo, se logró reducir la exigencia a la simple consistencia. La demostración del teorema se puede encontrar por ejemplo en (Hamilton, 1981), (Mosterín, 1981) y (Martínez & Piñeiro, 2009). Una idea general de la prueba es la siguiente. Se construye un enunciado G de un método axiomático P (básicamente la unión de la lógica de *Principia Mathematica* con la *axiomática de Peano*), tal que ni G ni ¬G pueden ser teoremas de P, porque de lo contrario se llega a una contradicción. Para construir G se emplea un ingenioso procedimiento que codifica la metateoría de P, conocido como *gödelización*, que asigna biunívocamente números naturales (*números de Gödel*) a los símbolos, las fórmulas y las sucesiones de fórmulas. De esta manera se logra transformar enunciados acerca de P en enunciados acerca de los números, y así expresarlos dentro del método. Los nuevos enunciados dan lugar a relaciones aritméticas recursivas, que se prueban expresables en P. En otras palabras, se logra expresar en el lenguaje de la aritmética, utilizando solo las operaciones de suma y multiplicación, propiedades de los enunciados como relaciones entre números. Un ejemplo es la relación aritmética Dem(m, n), que se cumple si y solo si m es el número de

Gödel de una sucesión de fórmulas que conforma una demostración de una fórmula cuyo número de Gödel es n. Mediante construcciones como ésta se alcanza de una manera bastante simple el enunciado G, interpretado de la siguiente manera: para todo número natural n, n no es el número de Gödel de una demostración en P de G. Así, G dice de sí mismo que no es demostrable (hay una analogía con la *paradoja del mentiroso*). Por último, asumiendo tanto que G es un teorema como que ¬G es un teorema, se llega a una contradicción.

Es interesante destacar que la argumentación de Gödel puede desarrollarse a partir de un único hecho matemático, la existencia en la aritmética de una operación de *concatenación*. Y que añadiendo G como axioma no se soluciona la incompletitud, sino que se mantiene el problema ahora con otro enunciado indecidible. Más en general, se formula que cualquier extensión recursiva sigue siendo incompleta. Otras axiomáticas incompletas son la *aritmética de segundo orden* y la *teoría de conjuntos de Zermelo-Fraenkel*. Un ejemplo de una axiomática completa es la aritmética *aditiva* o *de Presburger*, que no tiene la operación de multiplicación. Otro caso en este último sentido es la *teoría de primer orden de los números complejos*. La teoría es completa a pesar de que los números naturales son un subconjunto de los complejos, lo que no es contradictorio porque con enunciados de primer orden no puede definirse en el método la propiedad "ser natural", es decir que los números naturales "están" pero no pueden identificarse.

Incompletitud e indecidibilidad

La demostración del teorema de incompletitud de Gödel fue puramente lógica, no consideró la noción de *computabilidad* que recién llegó en 1936 fundamentalmente con A. Church y A. Turing. Así no se percibió a simple vista el problema general de la indecidibilidad en las lógicas. La indecibilidad es una razón más profunda, de la que se infiere la incompletitud como puede verse en lo que sigue. Si M es una máquina de Turing (Turing, 1936) que acepta la cadena vacía λ, lo hace mediante una computación con configuraciones de no más de m símbolos. Sea i un número natural codificado en notación binaria que representa una computación de M con configuraciones de longitud m. Se sabe que el enunciado que establece que M acepta λ es indecidible. Dicho enunciado se puede expresar con la fórmula $\exists i\ \exists m\ E_m(i)$, siendo E_m un predicado verdadero si y solo si i es el código de una computación que acepta λ con configuraciones de no más de m símbolos. De esta manera, por propiedad de las reducciones de problemas, la indecibilidad del problema referido en términos de máquinas de Turing implica la indecidibilidad en la aritmética, no hay algoritmo posible para decidir todos sus enunciados verdaderos. Notar en particular que la completitud en la aritmética implica su decidibilidad: dado un enunciado cualquiera φ, se ejecutan en paralelo dos máquinas de Turing M_1 y M_2, guiadas por los axiomas y reglas de inferencia, con el propósito de demostrar φ y ¬φ, respectivamente, aceptándose si M_1 acepta, y rechazándose si M_2 acepta.

Entre las lógicas de primer orden, una si no la más estudiada es el *cálculo funcional puro*, identificado comúnmente con F_0. La importancia de F_0 proviene del resultado que establece que si es decidible, también lo es cualquier otra lógica de primer orden. El problema de la decidibilidad en F_0, denominado oportunamente *Entscheidungsproblem*, fue declarado por Hilbert como el problema central de la lógica matemática. Se buscó intensamente una solución positiva, y algunos resultados preliminares alimentaron la creencia errónea de que el problema era decidible (por ejemplo Gödel había demostrado en 1929 que todas las fórmulas válidas son demostrables). Pero en 1936 Church y Turing demostraron lo contrario, en forma independiente, lo que constituyó otro golpe (el primero fue el de Gödel en 1931 con su teorema de incompletitud) para el programa de Hilbert. Church utilizó la técnica del λ-cálculo y Turing las máquinas que llevan su nombre. (El lector interesado se puede remitir a (Rosenfeld & Irazábal, 2013) para encontrar una demostración, desarrollada por Church, de la indecibilidad de la lógica de primer orden, mediante una reducción de problemas a partir de un problema clásico de la computabilidad.)

Incompletitud y aleatoriedad

G. Chaitin también encara la incompletitud por el lado de la algorítmica para deducir los límites del razonamiento formal (Chaitín, 2015). Lo hace ya no desde la indecidibilidad sino desde la *aleatoriedad*, en el marco de la *teoría algorítmica de la información*, la cual trata la complejidad computacional de una manera alternativa. Se centra en que el tamaño de una teoría es el del conjunto de los axiomas que la definen, o lo que es lo mismo, del programa o máquina de Turing que la genera. Establece que "comprensión es compresión". A través de un camino alternativo al de Gödel, Church y Turing, llega a la conclusión de que no puede existir una teoría de *toda* la matemática, la matemática tiene una complejidad *infinita*, no es completamente *compresible*. Más aún, suponiendo que se pudiera definir una axiomática completa, igual no se lograría demostrar que es la más concisa de todas. Si un programa P es más grande que un programa Q que simula un método axiomático, entonces con Q no se puede demostrar que P es *elegante*, es decir que es el programa más chico entre todos los programas que hacen lo mismo.

El caso particular que presenta Chaitín es su famosa constante Ω, definida como la probabilidad de que una máquina de Turing elegida al azar termine. La parte decimal de la constante se demuestra *aleatoria*, en el sentido de que es irreducible, algorítmicamente incompresible. Ω es un número perfectamente definido, la sumatoria $\Sigma_p \in P\ 2^{-|p|}$, siendo P el conjunto de todas las máquinas que paran, y |p| el tamaño de p. Ω no puede ser calculado, porque si lo fuera el *halting problem* sería decidible. De esta manera, demuestra que un método axiomático solo puede determinar tanta información de Ω como la que permite su propia complejidad, no más, por lo que la única manera de especificar más información de Ω consiste en introducir dicha información directamente en los axiomas. La matemática porta

información irreducible, hay incompletitud, el mundo de la verdad matemática tiene una complejidad infinita, mientras que todo método axiomático tiene complejidad finita.

Sensatez, completitud e interpretaciones

Solo a los efectos de simplificar la presentación de los métodos de verificación de programas es que hemos tratado en todos los casos únicamente con la interpretación de los números enteros, en realidad la interpretación estándar de los números enteros, habitualmente denotada con I_0. Naturalmente, lo deseable es que los métodos tengan el mayor alcance posible. Notar de todos modos que las pruebas de sensatez de los métodos de verificación de correctitud parcial que hemos desarrollado no dependen de la interpretación considerada. Efectivamente, los métodos tienen sensatez *total*, son sensatos para todas las interpretaciones (en el caso del método O para los programas concurrentes, que utiliza proof outlines, también se habla de sensatez *fuerte*). Por ejemplo, la fórmula de correctitud parcial $\{x \geq 0 \wedge y > 0\}\ S_{div}\ \{x = y\ .\ c + r \wedge r < y \wedge r \geq 0\}$ probada previamente, siendo S_{div} un programa de PLW que calcula la división entera, también es verdadera considerando el dominio de los números reales, un conjunto finito de números naturales $\{n \mid n < max\}$, etc. La notación correspondiente para el método H es la siguiente (lo mismo aplica para D y O):

$$Tr\ _I \mid\!\!-_H \{p\}\ S\ \{q\} \rightarrow\ \mid\!=_I \{p\}\ S\ \{q\}, \text{ para toda interpretación } I$$

Tr I contiene todas las aserciones verdaderas con respecto a la interpretación I. Que se cumpla $\mid\!=_I \{p\}\ S\ \{q\}$ significa que {p} S {q} es verdadera considerando I. Como en este caso {p} S {q} es verdadera para todas las interpretaciones, se dice que la fórmula es *válida*.

Con la sensatez de los métodos de verificación de correctitud total la cuestión es distinta por la utilización de las funciones cota, que tienen que estar definidas estrictamente dentro del dominio semántico de los números naturales. Es decir, la interpretación asociada a las funciones cota no tiene por qué coincidir con la interpretación asociada a la computación del programa considerado. Por ejemplo, sea el siguiente programa PLW:

$$S_{eps} :: \text{while } x > epsilon \text{ do } x := x / 2 \text{ od}$$

Supongamos que las variables *x* y *epsilon* son de tipo real, inicialmente mayores que cero. Claramente se cumple $\langle x = X\rangle\ S_{eps}\ \langle true\rangle$, porque los distintos valores positivos de x constituyen iteración tras iteración una secuencia decreciente estricta, y epsilon > 0. Una adecuada función cota para este caso podría ser:

$$\text{if } x > epsilon \text{ then } \lceil \log_2 X/epsilon \rceil - \log_2 X/x \text{ else } 0 \text{ fi}$$

Así las cosas, ahora lamentablemente la propiedad deseada de sensatez total no se cumple, es decir, a diferencia de la implicación de antes, no vale la siguiente:

$$Tr_{I} \vdash_{H^*} \langle p \rangle\ S\ \langle q \rangle \rightarrow\ \models_{I} \langle p \rangle\ S\ \langle q \rangle, \text{ para toda interpretación } I$$

Y lo mismo sucede con los métodos D* y O*. Como contraejemplo podemos mostrar simplemente el programa S :: while x > 0 do x := x – 1 od. Mediante el método H* se prueba fácilmente la fórmula ⟨true⟩ S ⟨true⟩, y si bien interpretado en el modelo *estándar* de los números naturales S siempre termina, no lo hace interpretado en un modelo *no estándar* (en el que a la sucesión infinita inicial asimilable a los números naturales le sigue un conjunto de cadenas de números no estándar, cada una de ellas sin mínimo ni máximo) cuando el valor inicial de la variable x es un número no estándar. Lo que se suele hacer en este caso es extender mínimamente: (a) el lenguaje de especificación con el de la axiomática de Peano (la habitual de los números naturales) más un predicado unario N que denota la propiedad "ser natural", y correspondientemente (b) la interpretación I considerada con el dominio de los números naturales estándar y sus operaciones aritméticas habituales. A la interpretación obtenida a partir de I se la conoce como *aritmética*, y se la denota con I^+. Se prueba de manera análoga a la que vimos previamente que:

$$Tr_{I+} \vdash_{H^*} \langle p \rangle\ S\ \langle q \rangle \rightarrow\ \models_{I+} \langle p \rangle\ S\ \langle q \rangle, \text{ para toda interpretación } I^+$$

Esto también aplica a los métodos D* y O*. En este caso se habla de sensatez *aritmética*. De esta manera ahora el problema con el programa S anterior desaparece, porque se puede plantear la prueba en H* de la fórmula de correctitud total ⟨N(x)⟩ S ⟨true⟩, para que los elementos no estándar que causan la no terminación no satisfagan la precondición.

En lo que hace a la completitud de los métodos de verificación de programas, la clase más básica que se puede plantear es la completitud *semántica*, que no hemos analizado por enfocarnos en una estricta noción de prueba sintáctica. La completitud semántica se refiere solamente a la existencia de conjuntos de estados en lugares determinados, ignorando la manera de expresarlos en un lenguaje de especificación, e incluso de construirlos algorítmicamente.

Cuando tratamos la completitud de los métodos de verificación de correctitud parcial introdujimos el concepto de completitud *relativa*. Relativa por un lado al conjunto de todas las aserciones verdaderas de la interpretación I utilizada cuando la axiomática asociada es incompleta (el caso por ejemplo de los números naturales y los números enteros). Y por otro lado relativa a la expresividad del lenguaje de especificación con respecto al lenguaje de programación y la interpretación I, que se cumple cuando se puede expresar la postcondición más fuerte de cualquier precondición p y cualquier programa S, es decir una aserción que denota el conjunto de estados $\{\sigma' \mid \exists\sigma: \sigma \models p \wedge val(\pi(S, \sigma)) = \sigma' \neq \bot\}$ considerando I, tal como se definió en la descripción del método H. A este tipo de completitud también se la conoce

como completitud *en el sentido de Cook*. Como se indicara previamente, hay una definición equivalente teniendo en cuenta la precondición más débil. Independientemente de habernos referido solamente a los números enteros estándar, notar que de la prueba de la completitud relativa del método H utilizando PLW que desarrollamos (y la del método D utilizando GCL), efectivamente se deduce que para todo lenguaje de especificación y toda interpretación I, si el lenguaje es expresivo con respecto a PLW (o a GCL) e I, entonces:

$$|=_I \{p\}\ S\ \{q\} \rightarrow Tr_I \vdash_H \{p\}\ S\ \{q\}$$

Una típica estructura a utilizar para que las especificaciones basadas en la lógica de primer orden sean expresivas es la axiomática de Peano, ya referida en un par de ocasiones. Es una estructura muy rica que permite codificar con números naturales computaciones de programas (por ejemplo recurriendo al procedimiento de *gödelización*), y así expresar con objetos del dominio las postcondiciones más fuertes. Como contraejemplo, la aritmética de Presburger también ya mencionada, que no tiene la multiplicación, no sirve para la expresividad requerida.

Para el tratamiento de la completitud de los métodos de verificación de correctitud total, al igual que en el caso de la sensatez se suele recurrir a las interpretaciones aritméticas. El método H* (también D* y O*) tiene completitud *aritmética*:

$$|=_{I+} \langle p\rangle\ S\ \langle q\rangle \rightarrow Tr_{I+} \vdash_{H^*} \langle p\rangle\ S\ \langle q\rangle, \text{ para toda interpretación } I^+$$

La interpretación aritmética también debe considerarse en el caso de la completitud del método O, para que puedan expresarse las historias de las computaciones que emplean variables auxiliares.

Naturalmente el objetivo a perseguir es contar con métodos de prueba sensatos, completos (de alguna manera) y composicionales, cualquiera sea la interpretación y el paradigma de programación considerados. Lamentablemente esto no puede garantizarse, en particular pero no solamente en el contexto de la concurrencia, en que las propias aserciones llegan necesariamente a tener estructuras muy complejas.

Composicionalidad

La importancia de la composicionalidad en un método de verificación radica en que permite probar (y construir) modularmente un programa, favoreciendo la escalabilidad y así la factibilidad de la aplicación del método. Vimos que la concurrencia constituye un desafío para los métodos composicionales. En concreto, el método O de correctitud parcial para los programas concurrentes SVL descripto previamente no es composicional. Si bien respeta la estructura modular de los programas, se deben establecer aserciones sobre estados intermedios no capturadas por el tipo de especificaciones que consideramos.

La misma autora, S. Owicki, planteó una variante de dicha aproximación que facilita el chequeo de las posibles interferencias (Owicki, 1976), considerando un lenguaje similar a SVL denominado RVL (por *resource variables language* en inglés, es decir lenguaje de recursos de variables). RVL ofrece una primitiva de sincronización que fuerza automáticamente la exclusión mutua de las secciones críticas, de la forma:

with r_i when B do S endwith

tal que r_i es un *recurso*, B una expresión booleana y S un subprograma de PLW, siendo los recursos conjuntos disjuntos de variables. Las variables de los recursos solo pueden ser utilizadas por las sentencias with, y toda variable compartida modificable debe estar definida dentro de un recurso (de esta manera se pueden identificar claramente las secciones críticas de un programa). La semántica informal del with es la siguiente: cuando un componente secuencial S_k está por ejecutar una instrucción with r_i when B do S endwith, si r_i está libre y B es verdadera, entonces S_k puede ocupar el recurso, obtener el uso exclusivo del mismo y progresar en la ejecución del with. Recién cuando S_k completa el with libera el recurso, para que algún otro componente lo utilice. No hay manejo de prioridades sobre los recursos. Los with no son atómicos, y naturalmente pueden causar deadlock. El mecanismo descripto es una simplificación de los *monitores*. Un monitor es más sofisticado, permite establecer políticas de priorización de procesos, liberaciones temporarias, y la implementación de *tipos de datos abstractos*, encapsulando datos y operaciones. Una regla de prueba habitual para la instrucción with (regla WITH), asumiendo para simplificar un solo recurso r, es:

$$\frac{\{I_r \wedge p \wedge B\}\ S\ \{I_r \wedge q\}}{\{p\}\ \text{with } r \text{ when } B \text{ do } S \text{ endwith}\ \{q\}}$$

La aserción I_r es un invariante asociado al recurso r, que vale toda vez que el recurso está libre. Las variables de I_r están en r. La regla establece que si se cumple $\{p \wedge B\}$ S {q}, entonces también se cumple {p} with r when B do S endwith {q}, pero con la condición de que la ejecución de S preserve, a partir de $p \wedge B$, el invariante I_r. Para la composición concurrente se utiliza la meta-regla SCC (por *sección crítica condicional*):

$$\frac{\{p_1\}\ S_1\ \{q_1\},\ \dots\ ,\ \{p_n\}\ S_n\ \{q_n\},\ \text{proof outlines que utilizan } I_r}{\{I_r \wedge p_1 \wedge \dots \wedge p_n\}\ S_1 \parallel \dots \parallel S_n\ \{I_r \wedge q_1 \wedge \dots \wedge q_n\}}$$

tal que para todo $i \neq k$, el conjunto de las variables libres de p_i y q_i y el conjunto de las variables modificables de S_k son disjuntos, y las variables libres de I_r están incluidas en r. Ahora, en la segunda etapa de la prueba no se necesita chequear que las proof outlines sean libres de interferencia, porque sus aserciones se refieren únicamente a variables no

compartidas o a variables compartidas de solo lectura. La información de las variables compartidas modificables se propaga desde el comienzo hasta el final de las proof outlines a través del invariante I_r, por medio de las distintas aplicaciones de la regla WITH. Al comienzo vale I_r porque el recurso r está libre.

Una aproximación composicional para correctitud parcial en concurrencia con memoria compartida es la de C. Jones (Jones, 1981). En cada componente secuencial se consideran además de la pre y postcondición, dos nuevas expresiones: una *condición de fiabilidad* (*rely condition*) para especificar lo que el componente "espera" del entorno de ejecución, y una *condición de garantía* (*guarantee condition*) que expresa cómo influencia el componente en el entorno. Las condiciones se formulan independientemente de la implementación de los componentes, y la verificación incluye probar relaciones entre ellas (lo que garantiza un componente debe implicar lo que esperan los otros de él). Notar que de esta manera la complejidad de la prueba crece linealmente con respecto a la cantidad de componentes, ya no exponencialmente como en la aproximación anterior. Detallando un poco, el método conocido como "rely-guarantee" consiste en lo siguiente, considerando para simplificar solo dos subprogramas secuenciales S_1 y S_2. Dados S_1 y S_2, una precondición p y una postcondición q para $S_1 \| S_2$, y un conjunto de variables auxiliaries A, se deben definir dos tuplas (p_1, q_1, $rely_1$, $guar_1$) y (p_2, q_2, $rely_2$, $guar_2$) para S_1 y S_2, respectivamente, tal que p_1, q_1, p_2 y q_2 son aserciones que como siempre definen conjuntos de estados, y $rely_1$, $guar_1$, $rely_2$ y $guar_2$ son relaciones binarias que definen transformaciones entre las variables compartidas y auxiliares. El significado de la tupla para S_1 es que, cuando S_1 se ejecuta a partir de un estado que satisfice p_1, en un entorno que puede cambiar las variables compartidas y auxiliares según $rely_1$, el componente lleva a cabo transformaciones respetando $guar_1$, y si termina lo hace en un estado que satisface q_1. Lo mismo aplica para S_2. De este modo, si las pruebas locales de S_1 y S_2 satisfacen las condiciones establecidas (en particular, si los invariantes de cada S_i se preservan considerando su condición $rely_i$), entonces se logra probar $\{p_1 \wedge p_2\}\ S_1 \| S_2\ \{q_1 \wedge q_2\}$, aplicando en los últimos pasos la regla que descarta las variables auxiliares, y la prueba de que $p \longrightarrow p_1 \wedge p_2$, y $q_1 \wedge q_2 \longrightarrow q$.

Aproximaciones como la de Jones no hacen obsoletas aproximaciones como la de Owicki. En general, los métodos no composicionales son más fáciles de utilizar que los composicionales, particularmente en el caso de los programas concurrentes con variables compartidas (por ejemplo, se torna menos dificultoso expresar los invariantes). De hecho hay pocos ejemplos de programas no triviales verificados con una técnica composicional.

Composicionalidad con lógica temporal

Hemos descripto un método de verificación de programas reactivos especificados con lógica temporal alineado con la aproximación global: ignora la estructura de los programas, trata directamente las computaciones como un todo. Un conocido enfoque composicional para la verificación de distintas propiedades y el desarrollo sistemático de programas concurrentes con variables compartidas se basa en el lenguaje UNITY (por *unbounded non deterministic iterative*

transformations en inglés, es decir transformaciones iterativas no determinísticas y no acotadas). El método se presentó en (Chandy & Misra, 1988), y se lo considera fundacional para la construcción de programas concurrentes correctos, con una relevancia comparable a la del método introducido por E. Dijkstra una década atrás para programas secuenciales. Los programas son de entrada/salida o reactivos, se especifican utilizando lógica temporal, no tienen flujo de control, y las instrucciones son asignaciones que se ejecutan de una manera no determinística hasta que eventualmente no hay más cambios en los estados, convergiéndose en lo que se conoce como *punto fijo*.

Las propiedades que se consideran en un programa UNITY son:

- p *unless* q. Si se cumple p alguna vez entonces q nunca se cumple y p se cumple para siempre, o bien q se cumple a futuro y p se cumple al menos hasta que q se cumple. Dos casos particulares son *stable* p e *invariant* p. El primer caso establece que si se cumple p alguna vez entonces se cumple para siempre, y el segundo caso que p vale siempre y desde el inicio. Estas son las propiedades safety.
- p *ensures* q. Si se cumple p alguna vez entonces p sigue valiendo mientras no se cumpla q, y a futuro se cumple q.
- p *leads-to* q. Una vez que se cumple p, se cumple q o a futuro se cumple q. Nada se puede decir acerca de si p sigue valiendo mientras no se cumpla q. Esta propiedad y la anterior son las propiedades de progreso.

El método se basa en dos estrategias de composición, la *unión* y la *superposición*. La unión de dos programas, que pueden compartir variables, es simplemente la concatenación de ámbos. La unión de P_1 con P_2 se denota con P_1 || P_2. Se demuestra que:

- Se cumple p unless q en P_1 || P_2 si se cumple p unless q en P_1 y se cumple p unless q en P_2.
- Se cumple p ensures q en P_1 || P_2 si se cumple p ensures q en un programa y p unless q en el otro.

Lamentablemente, no se puede afirmar que p leads-to q se cumple en P_1 || P_2 aún valiendo en los dos programas. De todos modos, el método propone un mecanismo general que permite inferir propiedades en P_1 || P_2 a partir de propiedades de los programas, lo que se conoce como propiedades *condicionadas*.

Por su parte, la superposición no es más que la estrategia de programar por capas. A una nueva capa se le agregan variables y asignaciones que no alteran el cómputo de las capas inferiores. De esta manera, entre la unión y la superposición se cubren las estrategias de descomposición y refinamiento, respectivamente, para el desarrollo sistemático de programas. La ausencia de una noción de descomposición en la superposición limita su tratamiento algebraico. La estrategia exige en general un conocimiento íntimo del programa subyacente. A

favor de la superposición se tiene entre otras cosas que facilita probar propiedades, y naturalmente que preserva las propiedades de las capas inferiores al no alterar sus cómputos.

Lógicas temporales

A. Pnueli fue quien primero consideró la lógica temporal para la verificación de programas (Pnueli, 1977). La semántica asignada a los programas era la que utilizamos en nuestro capítulo, *lineal*, según la cual un programa es el conjunto de todas sus computaciones, siendo una computación una secuencia de estados generados a partir de la ejecución de instrucciones atómicas, una por vez (la ejecución concurrente de los procesos la representamos por la intercalación de las mismas, incluyendo algún tipo de fairness que evita determinadas intercalaciones "injustas"). Una de las debilidades de esta aproximación es que omite la información de dónde en un programa se elige una entre varias posibilidades. Por ejemplo, si *or* representa como vimos antes una selección no determinística, el programa S ; (T or U) resulta semánticamente equivalente al programa (S ; T) or (S ; U), si bien en el primer caso se decide entre T o U después de la ejecución de S, mientras que en el segundo caso se decide antes. Otra debilidad es la imposibilidad de distinguir entre el no determinismo causado por una selección no determinística y el introducido por la semántica de intercalación para la concurrencia. Por ejemplo, (S ; T) or (T ; S) y S || T resultan ser programas semánticamente equivalentes a pesar de sus claras diferencias.

Una alternativa a la semántica lineal es la semántica *ramificada* (*branching* en inglés). En este caso un programa es un árbol de estados, cada nodo n representa un estado σ y los hijos de n son todos los estados σ' que pueden suceder a σ por la ejecución de una instrucción atómica. Siguiendo con los ejemplos anteriores, ahora se puede distinguir entre los programas S ; (T or U) y (S ; T) or (S ; U), pero por la semántica de intercalación, (S ; T) or (T ; S) y S || T siguen siendo semánticamente equivalentes.

Una semántica para la concurrencia que sí resuelve las dos debilidades referidas es la *semántica de orden parcial*, para la que un programa es una estructura de estados con dos relaciones, una de *precedencia* y otra de *conflicto*. Dos estados (y acciones) están en conflicto si no pueden integrar una misma computación. Por ejemplo, S y U están en conflicto en (S ; T) or (U ; V). La relación de conflicto se extiende por la relación de precedencia: en el mismo programa, S y U preceden a T y V, respectivamente, por lo que cada elemento de {S, T} está en conflicto con cada elemento de {U, V}. Dos estados (y acciones) no afectados por precedencia ni conflicto se consideran *independientes*: pueden ser ejecutados en paralelo. Así, esta semántica identifica la concurrencia como un fenómeno distintivo, no traducible a una intercalación. Por ejemplo, en S || T, S y T son independientes, mientras que en (S ; T) or (T ; S) existen cuatro componentes, digamos S_1, T_1, S_2 y T_2, que cumplen $S_1 < T_1$, $T_2 < S_2$, y cada elemento de $\{S_1, T_1\}$ está en conflicto con cada elemento de $\{T_2, S_2\}$; claramente estos dos programas tienen estructuras distintas. De la misma manera, S ; (T or U) difiere de (S ; T) or (S

; U): por ejemplo, la estructura correspondiente al segundo programa tiene dos componentes S_1 y S_2 en conflicto.

Las distintas semánticas mencionadas se corresponden con distintas lógicas temporales, lineal, ramificada y lógica sobre órdenes parciales. Acerca de la primera ya profundizamos en la segunda parte del capítulo. Diremos algo más sobre ella, y también consideraremos aspectos de la lógica ramificada, en ambos casos limitándonos a lo proposicional.

Lógicas LTL, CTL y CTL*

LTL (por *linear-time temporal logic* en inglés, es decir lógica temporal lineal) es una conocida lógica temporal lineal con operadores de futuro y cuyas fórmulas expresan propiedades sobre computaciones infinitas. Una fórmula φ de LTL tiene la siguiente sintaxis:

$$\varphi :: T \mid F \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \rightarrow \varphi \mid X\varphi \mid G\varphi \mid F\varphi \mid \varphi\ U\ \varphi \mid \varphi\ U_w\ \varphi \mid \varphi\ R\ \varphi$$

p es una proposición atómica perteneciente a un conjunto PA, y T y F son las constantes que representan los valores verdadero y falso, respectivamente. La semántica de LTL se define sobre sistemas de transiciones de la forma M = (Σ, ⟶, L), donde Σ es un conjunto de estados, ⟶ es una relación binaria en Σ tal que para todo estado σ existe otro estado σ' que cumple σ ⟶ σ', y L es una función L: Σ ⟶ P(PA) que asigna a todo estado un conjunto de proposiciones atómicas del conjunto PA. La definición semántica la hemos definido prácticamente toda previamente, salvo el caso de las constantes, las proposiciones atómicas y el operador R. Se define, dada una computación $\pi = \sigma_0, \sigma_1, \ldots$:

- $(\pi, i) \models T$
- $(\pi, i) \not\models F$
- $(\pi, i) \models p \leftrightarrow p \in L(\sigma_i)$
- $(\pi, i) \models \varphi_1 R \varphi_2 \leftrightarrow$ (a) existe algún $j \geq i$ tal que $(\pi, j) \models \varphi_1$, y para todo k, siendo $i \leq k \leq j$, se cumple $(\pi, k) \models \varphi_2$, o bien (b) para todo $j \geq i$, se cumple $(\pi, j) \models \varphi_2$. R es el operador *release* (libera). Difiere de U_w por permutar los roles de φ_1 y φ_2, y porque ahora en la parte (a) k alcanza a j.

Se puede reducir la cantidad de operadores temporales de LTL sin alterar su expresividad: se prueba que cualquiera de los conjuntos {U, X}, {U_w, X} o {R, X} es adecuado. Además, añadir operadores de pasado no agrega poder expresivo. Esto se puede explicar porque los operadores de pasado permiten recorrer hacia atrás una computación para alcanzar estados que pueden alcanzarse recorriendo hacia adelante la computación desde su inicio.

Como las fórmulas de LTL se evalúan sobre computaciones, se define que a partir de un estado σ se cumple una fórmula si *todas* las computaciones a partir de σ la satisfacen. Así, las

fórmulas de LTL están implícitamente cuantificadas universalmente, por lo que no pueden expresar propiedades sobre *algunas* computaciones. Este problema se puede resolver considerando la negación de una fórmula: existe *una* computación que satisface φ si y solo si *no todas* las computaciones satisfacen $\neg\varphi$. Pero esto solo es posible cuando una propiedad se puede formular sin combinar cuantificadores universales y existenciales.

CTL (por *computation tree logic* en inglés, es decir lógica sobre un árbol de computaciones) es una conocida lógica temporal basada en una semántica ramificada y resuelve el problema referido. Además de los operadores de futuro X, G, F y U, utiliza los cuantificadores A y E, universal y existencial, respectivamente. Una fórmula φ de CTL tiene la siguiente sintaxis:

$$\begin{aligned}\varphi ::\quad & T \mid F \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \rightarrow \varphi \mid \\ & AX\varphi \mid EX\varphi \mid AG\varphi \mid EG\varphi \mid AF\varphi \mid EF\varphi \mid A(\varphi\ U\ \varphi) \mid E(\varphi\ U\ \varphi)\end{aligned}$$

Todo operador temporal de CTL tiene dos símbolos, el primero de los cuales es un cuantificador. Los operadores U_w y R no están incluidos en el lenguaje, se pueden derivar a partir de los existentes. La semántica de los operadores AX y EX se define de la siguiente manera:

- $(\pi, i) \models AX\varphi \leftrightarrow$ para todo estado σ' tal que $\sigma_i \rightarrow \sigma'$, se cumple $\sigma' \models \varphi$.
- $(\pi, i) \models EX\varphi \leftrightarrow$ para algún estado σ' tal que $\sigma_i \rightarrow \sigma'$, se cumple $\sigma' \models \varphi$.

De un modo similar se puede definir la semántica del resto de los operadores cuantificados. Como en el caso de LTL, también se puede reducir la cantidad de operadores de CTL sin alterar su expresividad: se prueba que alcanza con algún operador de {AX, EX}, algún operador de {EG, AF, AU}, y el operador EU, que juega un rol especial por la inexistencia de U_w y R. Pero ahora agregando operadores de pasado se incrementa el poder expresivo, dada la estructura arbórea asociada a la semántica de CTL. Otra característica importante del lenguaje, conocida como *punto fijo*, es que los operadores AG, EG, AF, EF, AU y EU se pueden expresar en términos de ellos mismos y de AX y EX, de una manera inductiva. Por ejemplo, vale $AG\varphi = \varphi \wedge AXAG\varphi$, y $E(\varphi_1\ U\ \varphi_2) = \varphi_2 \vee (\varphi_1 \wedge EXE(\varphi_1\ U\ \varphi_2))$.

CTL* es otra conocida lógica temporal, que combina la expresividad de la lógica lineal LTL y la lógica ramificada CTL. Elimina la restricción sintáctica de CTL con respecto a los cuantificadores A y E. Contiene fórmulas de estado y fórmulas de camino. Una fórmula de estado φ tiene la siguiente sintaxis:

$$\varphi :: T \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid A\psi \mid E\psi$$

La fórmula ψ es una fórmula de camino. Una fórmula de camino ψ tiene la sintaxis:

$$\psi :: \varphi \mid \neg\psi \mid \psi \wedge \psi \mid X\psi \mid G\psi \mid F\psi \mid \psi\ U\ \psi$$

La fórmula φ es una fórmula de estado.

CTL permite explícitamente cuantificar computaciones, por lo que en este sentido es más expresiva que LTL. Por ejemplo, se prueba que la fórmula AGEFp, que expresa que siempre se puede alcanzar un estado que satisfaga p (útil por ejemplo para encontrar deadlocks en protocolos), no es expresable en LTL. Otro ejemplo se relaciona con una instancia del problema de exclusión mutua: (a) Un conjunto de procesos $P_1, \ldots, P_n$ funcionan como *terminal servers*. (b) Un proceso P_i puede recibir o no un carácter, y así puede permanecer siempre en su sección no crítica SNC_i mientras otros procesos reciben y procesan caracteres, o puede recibir un carácter e intentar ingresar a su sección crítica SC_i (estado TRY_i). Para expresar esta propiedad, considerando a P_i en SNC_i, se puede usar la fórmula $EG(inSNC_i) \wedge EF(inTRY_i) \wedge A(G(inSNC_i) \vee F(inTRY_i))$, que se prueba no es expresable en LTL (notar que la fórmula candidata $G(inSNC_i) \vee F(inTRY_i)$ no sirve porque contempla el caso en que todas las computaciones satisfacen $F(inTRY_i)$ y ninguna satisface $G(inSNC_i)$).

Como contrapartida, a diferencia de LTL, CTL no permite entre otras cosas seleccionar un rango de computaciones para describirlas mediante una fórmula, por lo que en este sentido LTL es más expresiva que CTL. Por ejemplo, en LTL se puede expresar la propiedad que establece que en todas las computaciones en que se cumple p también se cumple q, mediante la fórmula $Fp \longrightarrow Fq$. Se prueba que en CTL esta propiedad no puede expresarse, y el motivo es la obligación de anteponer un cuantificador al operador F (la fórmula candidata $AFp \longrightarrow AFq$ no expresa lo mismo, tampoco $AG(p \longrightarrow AFq)$). Otro caso tiene que ver con el fairness: solo LTL permite expresarlo. Por ejemplo, la fórmula $GFp \longrightarrow Fq$, que establece que si p se cumple infinitas veces entonces también se cumple q, no es expresable en CTL.

CTL* es más expresiva que LTL y CTL. Por ejemplo, se prueba que la fórmula EGFp de CTL*, que expresa que existe una computación en que se cumple infinitas veces p, no es expresable en los otras dos lógicas.

La cuestión de cuál es la "mejor" de las lógicas en términos prácticos viene siendo desde hace mucho tiempo materia de debate. Lógicas temporales lineales como LTL son muy populares por su simplicidad. En particular se las consideran adecuadas para verificar programas concurrentes. Es natural en este contexto querer asegurar que una propiedad se cumpla en todas las computaciones. En cuanto a las actividades de especificación y diseño en cambio, la necesidad de expresar la existencia de computaciones alternativas puede motivar la preferencia por una lógica temporal ramificada, teniendo en cuenta el no determinismo presente en muchos programas concurrentes. Particularmente estas lógicas resultan útiles para la síntesis o derivación automática de programas, en que se requieren especificaciones muy precisas (se podría también recurrir a la lógica lineal, pero utilizando algunos artificios para asegurar la existencia de computaciones alternativas). La cuantificación explícita puede ayudar también a asegurar que un programa exhiba un adecuado grado de paralelismo, es decir que no degenere en un programa que ejecuta una sola computación. En definitiva, en general está aceptado que debería usarse el subconjunto de CTL* más apropiado para el propósito

buscado, teniendo en cuenta parámetros como la expresividad y la complejidad computacional de la prueba de satisfactibilidad.

Model checking

A lo largo de todo el capítulo nos hemos centrado en la verificación de programas conocida como *orientada a pruebas* (*proof oriented* o *theorem proving* en inglés), aproximación según la cual un programa o sistema y las propiedades a probar en él se expresan mediante fórmulas de una lógica. Las pruebas pueden construirse manualmente o con soporte herramental, este último habiendo progresado sustancialmente en los últimos años (cada vez es más profuso el uso de demostradores de teoremas automáticos o semi-automáticos para probar propiedades críticas en el hardware y software).

Hay otra aproximación, muy difundida, que es el *model checking* (del inglés: *chequeo o verificación de modelos*). Surgió en 1981 con el trabajo de E. Clarke y E. Emerson (Clarke & Emerson, 1981). El model checking consiste en construir un modelo finito de un programa o sistema y chequear que una determinada propiedad es satisfecha por el modelo. El chequeo se lleva a cabo mediante una búsqueda exhaustiva a lo largo del espacio de estados construido, que se garantiza que termina por la finitud del modelo. A diferencia de la verificación orientada a pruebas, esta aproximación es completamente automática y rápida, en algunas ocasiones produciendo una respuesta en cuestión de minutos. El desafío algorítmico radica fundamentalmente en el manejo de estructuras de datos muy grandes, lo que se conoce como el problema de la *explosión de estados*.

El model checking se basó inicialmente en especificaciones en lógica temporal, pero desde los años 1990 también considera el formalismo de los *autómatas de predicados*. Dichos autómatas contienen un conjunto de condiciones de transición que establecen cuándo se puede avanzar de un estado a otro, un subconjunto de estados recurrentes R, y un subconjunto de estados persistentes P. Las cadenas de entrada representan computaciones infinitas, cuyo primer estado coincide con el primer estado del autómata. Se define que un autómata de predicados acepta una computación, si en todas sus trazas el conjunto de estados que ocurre infinitas veces no es disjunto con R o bien está incluido en P.

Para verificar una fórmula φ en un programa especificado con una lógica como CTL, el algoritmo típico de model checking consiste, dado un sistema de transiciones o modelo M, en etiquetar los estados que satisfacen φ. Se basa en la propiedad de punto fijo mencionada en el apartado anterior. Se empieza con poblar los estados con las subfórmulas más pequeñas, y luego iterativamente se tratan las subfórmulas de mayor longitud, considerando todos los casos de los operadores, hasta llegar a la fórmula completa φ. El algoritmo es eficiente en términos de tiempo con respecto tanto al tamaño de φ como de M. Dada la imposibilidad de tratar sintácticamente el fairness, esta restricción debe establecerse directamente a través del algoritmo.

Para el caso de programas especificados con lógicas como LTL existen varios algoritmos de model checking, pero todos se basan en una misma estrategia general, tomando como base el formalismo de los autómatas de predicados pero combinado con el uso de fórmulas de lógica temporal. Con más detalle, para verificar que M satisface φ, es decir que φ se cumple en todas las computaciones de M, básicamente: (1) Se construye un autómata A para reconocer la fórmula ¬φ, es decir para aceptar todas las correspondientes secuencias de valuaciones de proposiciones atómicas que satisfacen ¬φ. (2) Se combina A con M, lo que produce entonces un sistema de transiciones cuyos caminos son tanto de A como de M. (3) El model checker acepta si y solo si no existe ningún camino en el sistema obtenido, porque entonces ninguna computación satisface ¬φ. Esta estrategia es más costosa que la que describimos para lógicas como CTL, lo que tiene sentido porque las fórmulas deben ser evaluadas en caminos en lugar de estados: en términos de tiempo es polinomial con respecto al tamaño de M y exponencial con respecto al tamaño de la fórmula.

Bibliografía

Apt, K. (1981). *Ten years of Hoare's logic, a survey, part 1*. ACM Trans. Prog. Lang. Syst., 3, 431-483.

Apt, K. & Olderog, E. (1997). *Verification of secuencial and concurrent programs, second edition*. Springer.

Chaitín, G. (2015). *El número omega: límites y enigmas de las matemáticas*. Tusquets Editores.

Chandy, K. & Misra, J. (1988). *Parallel program design: a foundation*. Addison-Wesley.

Clarke, E. & Emerson, E. (1981). *Synthesis of synchronization skeletons for branching time temporal logic*. Logic of Programs: Workshop (Yorktown Heights, NY). Vol. 131 of Lecture Notes in Computer Science. Springer-Verlag.

de Bakker, J. (1980). *Mathematical theory of program correctness*. Englewood Cliffs.

Francez, N. (1992). *Program verification*. Addison-Wesley.

Mosterín, J., ed. (1981). *Kurt Gödel. Obras completas*. Alianza Editorial.

Hamilton, A. (1981). *Lógica para matemáticos*. Paraninfo.

Huth, M. & Ryan, M. (2004). *Logic in computer science*. Cambridge University Press.

Jones, C. (1981). *Development methods for computer programs including a notion of interference*. PhD thesis, Oxford University Computing Laboratory.

Manna, Z. & Pnueli, A. (1989). *The anchored version of the temporal framework.* Proceeding linear time, branching time and partial order in logics and models for concurrency, school/workshop, 201-284. Springer-Verlag.

Manna, Z. & Pnueli, A. (1992). *The temporal logic of reactive and concurrent systems. Specifications*. Springer-Verlag.

Manna, Z. & Pnueli, A. (1995). *Temporal verification of reactive systems. Safety.* Springer.

Martínez, G. & Piñeiro, G. (2009). *Gödel ∀ (para todos).* Seix Barral.

Owicki, S. (1976). *A consistent and complete deductive system for the verification of parallel programs*. Proc. 8th Sym on Theory of Computation, Houston TX, October 1976.

Pnueli, A. (1977). *The temporal logic of programs*. Proceedings of the 18th Annual Symposium on Foundations of Computrer Science, IEEE, New York, 46-57.

Rosenfeld, R. & Irazábal, J. (2010). *Teoría de la computación y verificación de programas*. McGraw-Hill Educación, EDULP.

Rosenfeld, R. & Irazábal, J. (2013). *Computabilidad, complejidad computacional y verificación de programas.* EDULP.

Turing, A. (1936). *On computable numbers, with an application to the Entscheidungsproblem*. Proc. London Math. Society, 2(42), 230-265.

Los Autores

Pons, Claudia

Claudia Pons es Doctora en Ciencias Informáticas de la Universidad Nacional de La Plata (año 2000), y Especialista en Docencia Universitaria (Carrera Docente Universitaria), título otorgado por la misma Universidad en 2002. Ha sido Investigadora del Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), y actualmente es Investigadora Adjunta en la Comisión de Investigaciones Científicas de la Provincia de Buenos Aires (CICPBA). Desde 2010 es Directora del Centro de Altos Estudios en Tecnología Informática (CAETI) de la Universidad Abierta Interamericana (UAI), Ciudad Autónoma de Buenos Aires, Argentina. Su área de interés es la ingeniería de software, en particular el desarrollo de software dirigido por modelos, los lenguajes y metodologías de modelado de software y la verificación formal de programas.Es miembro del Centro de investigación LIFIA y ejerce la docencia en carreras de grado y posgrado en la Facultad de Informática de la Universidad Nacional de La Plata y en la UAI. Actualmente dirige proyectos de investigación y desarrollo de software y es autora de varios artículos científicos referidos al tema.

Rosenfeld, Ricardo

Ricardo Rosenfeld obtuvo el título de Calculista Científico de la Facultad de Ciencias Exactas de la Universidad Nacional de La Plata, Argentina, en 1983, y completó los estudios de la Maestría en Ciencias de la Computación del Instituto de Tecnología Technión, Israel, en 1991. Desde 1991 se desempeña como Profesor en la Universidad Nacional de La Plata, en las áreas de teoría de la computación y verificación de programas. Previamente, entre 1984 y 1990, fue docente en la Universidad Nacional de La Plata (lenguajes y metodologías de programación), en la Universidad de Buenos Aires (verificación y derivación de programas), en la Escuela Superior Latinoamericana de Informática (algorítmica y estructuras de datos, y teoría de compiladores), y en el Instituto de Tecnología Technión de Israel (programación). Publicó los libros *Teoría de la Computación y Verificación de Programas* (2010, EDULP, McGraw-Hill) y *Computabilidad, Complejidad Computacional y Verificación de Programas* (2013, EDULP). Es además uno de los Socios de Pragma Consultores, empresa regional dedicada a la tecnología de la información, ingeniería de software y consultoría de negocios.

Smith, Clara

Clara Smith es abogada de la matrícula, y Profesora de programación lógica en la Universidad Nacional de La Plata, luego de obtener su Doctorado por la misma Universidad, haber sido Becaria del CONICET y haber trabajado para la Suprema Corte de Justicia de Buenos Aires. En los años 2005 y 2006 obtuvo una beca del gobierno italiano para realizar una investigación en la Universidad de Bologna, Italia. En 2012 se desempeñó como Profesora Invitada en la Universidad de Toulouse 1 Capitole, Francia. En 2013 fue Directora de la carrera de Ingeniería en Sistemas de Información de la Universidad Católica de La Plata y diseñó el plan actualmente vigente de la Licenciatura en Sistemas de esa misma Universidad. En 2014 obtuvo una beca de la Unión Europea y ocupó una posición de staff en la Universidad de Bologna. En 2017 se desempeñó como Marie Skodovska Curie Fellow, también en Bologna. Sus temas de investigación se relacionan con los sistemas multiagente y la filosofía del derecho. Sus publicaciones científicas más importantes son *Modes of Adjointness*, con Matías Menni, y *Collective Trust and Normative Agents*, con Antonino Rotolo.

Pons, Claudia
Lógica para informática : lógica clásica, modal y de programas / Claudia Pons ; Ricardo Rosenfeld ; Clara Smith. - 1a ed . - La Plata : Universidad Nacional de La Plata, 2017.
Libro digital, PDF

Archivo Digital: descarga y online
ISBN 978-950-34-1510-8

1. Lógica. 2. programa. I. Rosenfeld, Ricardo II. Smith, Clara III. Título
CDD 005.16

Diseño de tapa: Dirección de Comunicación Visual de la UNLP

Universidad Nacional de La Plata – Editorial de la Universidad de La Plata
47 N.º 380 / La Plata B1900AJP / Buenos Aires, Argentina
+54 221 427 3992 / 427 4898
edulp.editorial@gmail.com
www.editorial.unlp.edu.ar

Edulp integra la Red de Editoriales Universitarias Nacionales (REUN)

Primera edición, 2017
ISBN 978-950-34-1510-8



FACULTAD DE
INFORMÁTICA



